UNIVERSITY OF OSLO
Department of informatics

Automating Weaving of Interaction Models in The Aspect Oriented Model Driven Framework using Kermeta

Master thesis

60 credits

Mansur Ali Abbasi
[mansuraa@ifi.uio.no]

1st August 2007

Preface

"In the Name of Allah, the Most Gracious, the Most Merciful"
- Quran: The Opening (Chapter 1), verse 1

This is a master thesis submitted to Department of Informatics, Faculty of Mathematics and Natural Sciences at University of Oslo in partial satisfaction of the requirements for the degree of Master of Science with specialization in Informatics (MSc Informatics).

The work reported in this thesis has been carried out at SINTEF Information and Communication Technology, Department of Cooperative and Trusted Systems, under the supervision of Arnor Solberg (PhD candidate and senior research scientist) and Dr Arne Jørgen Berre (associate professor and chief scientist) during the time period from August 2006 to July 2007.

The context of work has been AOMDF - the Aspect Oriented Model Driven Framework – a framework for model based, aspect oriented software design, developed through collaborative research activity between scientists at SINTEF ICT (Oslo, Norway) and Colorado State University (CO, USA). The results of this thesis are a direct enhancement to AOMDF and function as an important proof-of-concept and initial step on the path from the theoretical framework to a complete tool-supported methodology.

In my work with this thesis I have learned numerous lessons that I believe will prove important throughout my entire life. Firstly, I have learned the art of patience while analyzing and solving complex problems. Mastering this art not only yields correct comprehension of the problems and precise solutions but also provides room for applying creativity and thus introduce the dimension of elegancy in the final solutions. Secondly, I have achieved greater respect for the human mind and its cognitive potential by observing the continuous expansion of my own cognitive skills during this work and also the development of other people during my teaching appointments at the university. All other lessons are subsidiary to these two and I believe every young scientist and engineer will benefit from learning these lessons early in his or her career.

It is certainly up to the readers to decide and evaluate my mastery of the subject matter presented in this thesis, but hopefully this will be secondary to the contribution which I hope and believe this thesis brings to the ongoing research within the aspect oriented modeling community.

Acknowledgements

I would like to express my sincere thanks to my supervisors Arnor Solberg (senior scientist at SINTEF ICT) and Dr Arne Jørgen Berre (chief scientist at SINTEF ICT) who together have given me the opportunity to learn about and acquire hands-on experience with state-of-the-art research in software engineering. Their dedication to their work and vast technological overview has inspired me beyond my own expectations and introduced me to a whole new level of knowledge and learning.

Through my affiliation with SINTEF, I have had access to a large network of scientists, engineers, research fellows and students working on more or less related research around the globe. Some of them have played an important role as discussion partners. Among these I would especially like to thank the following:

- *Dr Øystein Haugen* (SINTEF, Norway) for sharing his expert insight into the inner workings of the UML metamodel and providing great motivation and guidance.
- *Dr Robert France* (Colorado State University, USA) for looking through parts of my work and indirect mentoring through Arnor Solberg.
- Research Fellow and PhD student *Jon Oldevik* (SINTEF, Norway) for sharing his research ideas and expert knowledge of Eclipse-based modeling tools.
- Research Engineer *Didier Vojtisek* (INRIA, France) for extensive support on the Kermeta language and workbench.
- *Dr Jacques Klein* (INRIA, France) for sharing parts of his work on scenario aspect weaving.
- Research Scientist *Gøran Olsen* (SINTEF, Norway) for sharing his insight into model transformations.

For assistance with practical matters, and for making me feel as a worthy part of the SINTEF research community, I wish to thank:

- *Bjørn Skjellaug* (Research Director)
- *Stine Holm* and *Anne Lund* (Project Secretaries)

Finally, I wish to thank my parents, my brother, my sisters and all my good friends for their outstanding, endless support, patience and encouragement throughout my studies and work with this thesis.

Table of Contents

List of Figures

List of Tables

1 Introduction

This chapter provides an initial introduction to the work reported herein by briefly introducing the context and motivation of the thesis and the research goals, scope and method. The document structure and relations between chapters are described in the thesis overview section.

1.1 CONTEXT OF WORK

The context of this work has been AOMDF - the Aspect Oriented Model Driven Framework. AOMDF [1-5] is a framework that describes a particular software design approach based on the general concepts of Aspect Oriented Modeling (AOM).

Aspect oriented modeling [6] is recognized as a state-of-the-art technique for analysis- and design-time identification, comprehension and modularization of crosscutting concerns in (distributed) software systems. Aspect oriented modeling is best understood when thought about as the result of a fusion between the advanced *separation of concerns* [7, 8] mechanisms offered by the two emerging software development paradigms *Model Driven Engineering (MDE)* [9-13] and *Aspect-Orientation (AO)* [14].

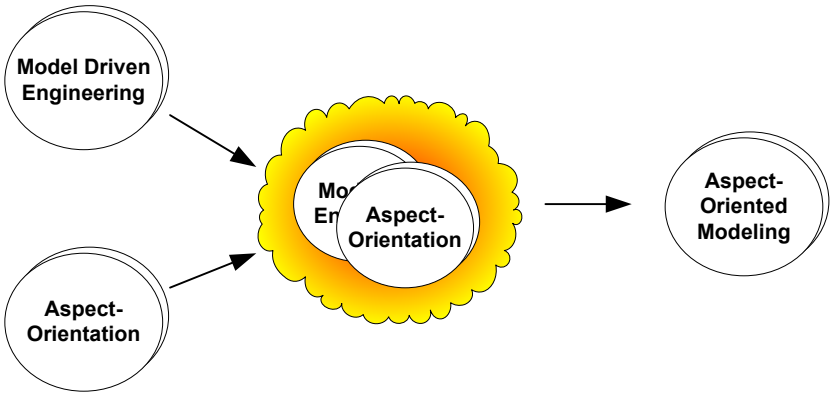

Figure 1-1: Fusion of Model Driven Engineering and Aspect-Orientation

While AOM is an umbrella concept covering multiple different approaches for model based, aspect-oriented software development, AOMDF is a specific approach centered only about the design-time separation and weaving of crosscutting concerns. Independent surveys have pulled forward AOMDF as one of the most mature and sophisticated among recently publicized, design-level AOM-approaches [15]. The main reason for this is the fact that AOMDF facilitates weaving of *crosscutting concerns* at the modeling (i.e. platform independent) level, while the majority of other approaches defer the weaving to the programming (i.e. platform specific) level [15-17].

Separation of Concerns, *Model Driven Engineering* and *Aspect-Orientation* are introduced in chapters 2, 3.4 and 3.5 respectively. Chapter 4 provides a brief introduction to *AOMDF*.

1.2 MOTIVATION

AOMDF covers structural and behavioral (visual) modeling of systems in an aspect-oriented fashion through minor suggested extensions to UML [18-20] class- and sequence diagrams (i.e. class- and interaction models). The main contribution of AOMDF is to enable separate, isolated design of the base, core features of a system and its crosscutting features (i.e. functionality that is spread across various modules of the system and tangled within other functionality). Using AOMDF, system developers may design the crosscutting features as reusable aspect models and the base features as primary models. The primary and aspect models are then woven (composed) together to obtain an integrated view of the complete system. The weaving can be carried out manually by experts, but due to the tediousness and error-proneness involved [21, 22], scalability of AOMDF requires automation of this process.

A recent case study presented in [23] has demonstrated how AOMDF can reduce the complexity involved in the design of distributed, adaptive systems with multiple crosscutting concerns. However, before extending and putting AOMDF to test in real world projects there is a need to enhance and strengthen the framework by providing proper proof-of-concept work on model weaving automation. Currently, only an initial proof-of-concept implementation for class model weaving exists [3, 24], but is still quite far away from functioning as a proper foundation for any kind of future tool-support. The part of AOMDF dealing with weaving of behavioral, interaction models [4] only exists at the theoretical level and lacks a convincing proof-of-concept foundation.

The existence of such a proof-of-concept foundation would directly enhance the current research on AOMDF along multiple dimensions: The practicality of the theoretical concepts would be justified, the level of tool-readiness and scalability would be raised, and new, valuable insight would be gained through the identification and handling of expected, as well as unexpected, challenges in the process for weaving interaction models.

1.3 RESEARCH GOALS, SCOPE AND METHOD

1.3.1 Goals and Scope

The main, overall goal of this thesis is, as motivated in the previous section, to design and construct a thorough proof-of-concept for weaving of interaction

models in AOMDF. Achievement of this goal will be of significant importance in exploring viability, technical issues and future work directions.

An ideal way to develop this proof-of-concept could have been to prototype a modeling toolkit with all the components shown in Figure 1-2. The Modeling Tool component would allow software developers to design base and aspect models and specify bindings between them, according to the proposed visual syntax of AOMDF. The Aspect Repository component would facilitate storage of reusable aspect models and the Model Weaver component would handle weaving of models using binding specifications and models loaded in the modeling tool as input. All components would depend on an AOMDF metamodel defining the constructs and rules needed to build and semantically process models in the AOMDF-domain.

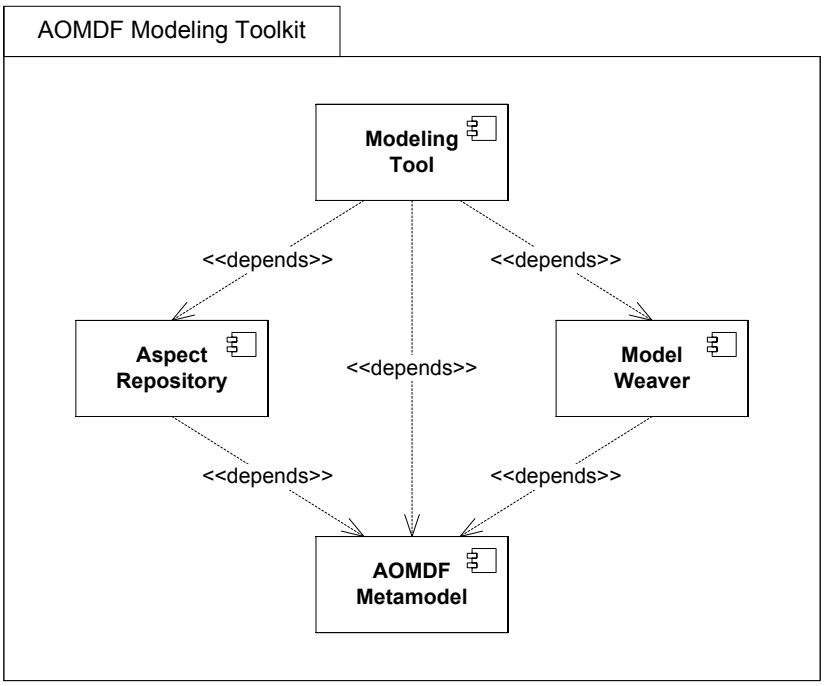

Figure 1-2: Components of a conceptual modeling toolkit for AOMDF

Developing the complete ideal prototype would of course be interesting at a later stage, but for now we limit ourselves to only focus on the metamodel and the model weaver as illustrated in Figure 1-3 on next page. These are the two components that will yield most research value, provide us with the insight we seek and strengthen the framework.

With this narrowing of scope, the main goal decomposes into the two following sub-goals:

- Develop a metamodel (i.e. an abstract syntax) for AOMDF interaction models
- Design and implement a model weaver for AOMDF interaction models

Further decomposition and scoping of these sub-goals, and discussion of related challenges, is addressed in chapter 8 (Technical Solution Approach) as

this requires the reader to have some insight into AOMDF (chapter 4) and the Technology Foundation (chapters 6, 7 and 7). The next subsection outlines the method employed to approach these goals.

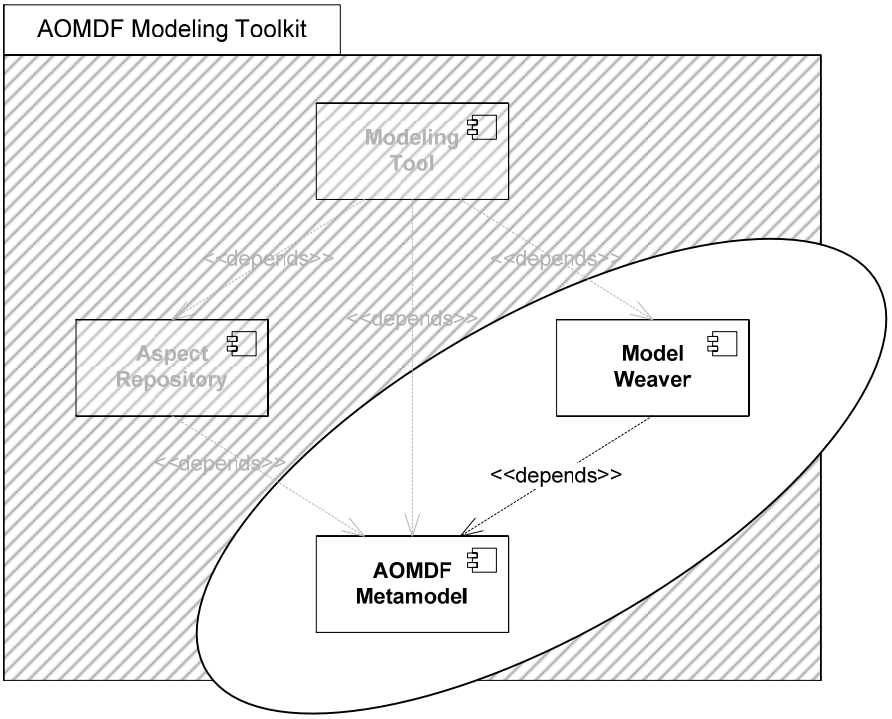

Figure 1-3: Scope of thesis

1.3.2 Method

With respect to ACM's taxonomy of computer science presented in [25], the work reported in this thesis falls into the subject area of Software Methodologies and Engineering, and is intertwined into the intersection of the theory, abstraction and design paradigms – although with most of it's weight in the latter. Hence, an experimental development method is suitable to approach the two sub-goals identified in the previous subsection.

We naturally divide the work into three steps as follows:

1.3.2.1 Step 1 – Develop Metamodel for AOMDF Interaction Models

The natural starting point is to define the constructs and rules needed for semantic processing of AOMDF models. Within the context of model driven engineering, the proper way to do this is by developing a metamodel [11, 12, 26, 27] – an object-oriented model defining what elements may exist within models in the AOMDF-domain and how these elements can be related to each other. In other words, we develop an abstract syntax [26, 27] for AOMDF interaction models.

One would assume that a suitable (partial) metamodel for AOMDF should already exists and may potentially be reused, but this is not the case. Previous

proof-of-concept efforts [3, 24] have focused solely on the class-model weaving part, and employed approaches that we find to be less suitable for describing interaction models. This leads us to an early decision to not directly extend the existing work, but rather start with developing a new metamodel which focuses on the interaction model part of AOMDF, and later try to integrate it with the existing work on class model weaving. We elaborate on the arguments for this decision in chapter 8 (Technical Solution Approach). The actual integration of our work with the previous work is left out of the scope this thesis.

Metamodelling is a complex engineering activity and mastery of it requires significant practice, skills developed over time, and great portions of creativity. In order to ensure that the metamodel we develop is of a high quality, we follow the metamodelling process suggested in [27].

As an input to this step, we take the recent papers [1-5] in which we find the concrete syntax [26, 27] proposed for AOMDF interaction models. The main output of this step is a metamodel – the abstract syntax model – for AOMDF interaction models. This will be the input to the next step. As a natural by-product, improvements – in the form of modifications – to the concrete syntax may result from the metamodelling activity.

For an early validation of the abstract syntax model, we will attempt to instantiate meaningful AOMDF interaction models based on our metamodel.

1.3.2.2 Step 2 – Design and Implement Model Weaver

A model weaver is a tool that automates model composition – the process of merging two or more models into a single, consistent and coherent model [21, 22]. In light of model driven engineering, model composition is a *model-to-model transformation* that takes multiple models as input and produces a single output-model. The design and implementation of the AOMDF interaction-model weaver is thus equivalent to defining an *endogenous horizontal transformation* [28]. (We return to the arguments for this classification in chapter 8 (Technical Solution Approach).

Several domain-specific languages and tools like [29-31] are suitable for our purpose, however weaving of interaction models is expected to involve several challenges related to the high degree of spatial information contained in behavior models. Thus, we choose to employ Kermeta [32], a metaprogramming and metamodel engineering workbench. The earlier mentioned work on class-model weaving [3, 24] also utilized Kermeta. However, Kermeta has evolved significantly and now offers new features and separation-of-concern aids (including aspect-orientation) and is well suitable for dealing with heavy-duty transformations. Kermeta is introduced in more detail in chapter 7.

The metamodel from step 1 will be the main input to this step. Thus, as soon as the metamodel reaches a certain level of completeness (i.e. it is not fully complete but describes most parts of the desired final metamodel in a well-formed manner), design and prototyping of the model weaver should begin.

This way we can iterate back and forth between step 1 and step 2 in an evolutionary fashion and carefully mold our metamodel into the desired final abstract syntax model, whilst ensuring good design of our model weaver.

1.3.2.3 Step 3 – Validation

The validation of the metamodel developed in step 1 and the interaction model weaver developed in step 2, is done informally by designing simple test cases (i.e. primary and aspect models), weaving them using our model weaver and evaluating the output models by means of manual model checking.

Successful modeling of the test cases will in itself validate the metamodel, while the weaver is validated upon successful weaving results.

1.3.2.4 Summary

Figure 1-4 summarizes our three-step method and illustrates its iterative and evolutionary nature.

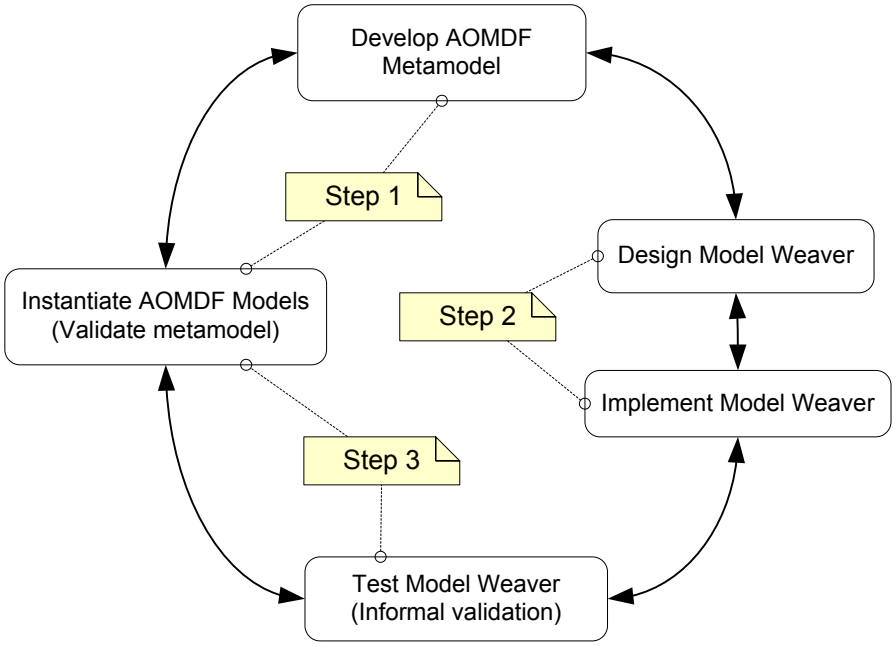

Figure 1-4: Method overview

1.4 THESIS OVERVIEW

The chapters of this thesis are divided into four parts as follows as shown in figure Figure 1-5 on the next page.

In the chapters of Part 1 (Background and Context) we present the background and context of our work. In the chapters of Part 2 (Technological Foundation) we present the fundamental concepts which the reader needs to understand in order to follow our work. In Part 3 (Contribution) we present our technical solution approach and the results of our work. In Part 4 (Discussion) we

finalize with a conclusion and future work ideas. We briefly also mention some related research.

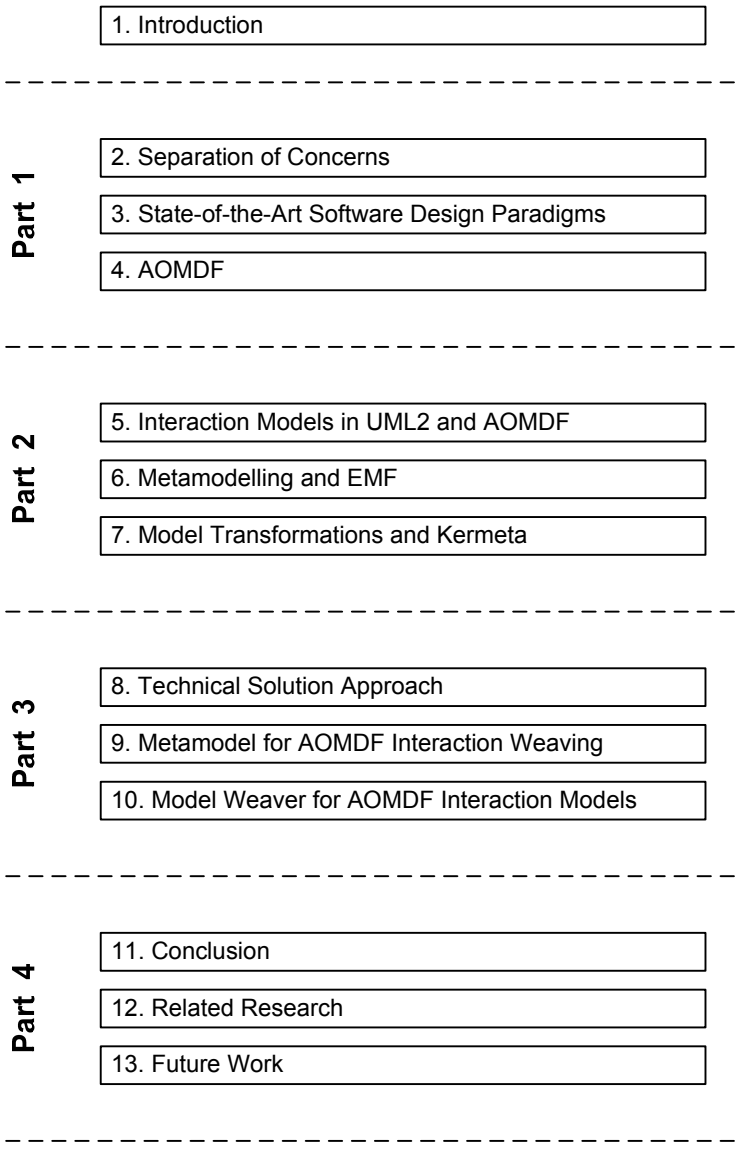

Figure 1-5: Thesis overview

PART I

BACKGROUND

AND

CONTEXT

---

2 Separation of Concerns in Software Design

In this chapter we introduce the principles *separation of concerns* (*SoC*) and *multidimensional separation of concerns* and their role in software design. We also provide a brief overview of well-established software design paradigms currently appraised by the industry.

2.1 THE SOC PRINCIPLE

There is no doubt that the design of software systems is a mentally complex and demanding activity. Software developers have to reason abstractly, use their creativity and think algorithmically in multiple dimensions all at the same time. The key to handling this successfully lies in being able to effectively order one's thoughts, and the only available technique of doing so is through *separation of concerns* as Edsger W. Dijkstra points out in his paper "On the role of scientific thought" [33].

Dijkstra explains the application of *separation of concerns* as the act of considering in depth only a single aspect of a certain subject matter in isolation for the sake of the aspect's own consistency, while knowing that one is occupying one's mind with only one among all the aspects of the whole subject matter, and hence being one- and multiple-track minded at the same time [33].

Certainly, although Dijkstra is often given the credit of coining this term, separation of concerns is a long standing idea, rooted in concepts like *divide and conquer*, and applicable not only in computer science but as a design and organization principle in many areas ranging from urban planning and architecture to management and mathematics. It simply means to break down a larger problem in sub-problems that are solved individually and the solutions combined to form the solution of the larger problem [8].

In the context of software design, separation of concerns is understood as the decomposition of a system into distinct pieces of interest or focus, i.e. into distinct concerns[1], that overlap as little as possible in functionality [8]. This is what lies at the heart of making the mental process of elegantly designing and analyzing large, complex systems comprehensible. Thus, the ultimate goal of every software design paradigm is to contribute with improved support for

[1] Concerns are, typically, synonymous with features or behaviors.

separation of concerns, and by doing so narrow the translation gap between requirements and design.

Traditional design approaches have contributed to improved separation of concerns by means of abstraction, encapsulation and modularization along a single – or a small set of – dimensions. However, the need for continuously improved, multi-dimensional separation of concerns seems to be ever-growing as we tend to build increasingly larger and complex systems.

The next section briefly introduces the major, well-established design paradigms currently appraised (more or less ubiquitously) by the mainstream software industry.

3 State-Of-The-Art Software Design Paradigms

3.1 OBJECT-ORIENTED DESIGN

Object orientation [34], with roots as long back as in the 1960s but no mainstream appraisal before in the mid-1980s, revolutionized the software industry by providing support for better modularity in software design and introducing concepts that since then have made it easier to map real world phenomena into software elements and vice versa.

The key design principle in object orientation is encapsulation by connecting data with its related operations and letting these serve as the system's data-access points. The system is in that way broken down into objects (components) with minimal interfaces and each of these can be viewed as an independent little machine with a distinct role and responsibility in the overall system.

Object orientation surely has contributed to improved separation of concerns. Despite its shortcomings and pitfalls, object orientation was a gigantic step in the right direction when introduced. However, modern software needs have now outgrown what object orientation alone may offer of concern separation; yet it remains the key software design paradigm which the majority of new approaches are bootstrapped upon.

3.1.1 Object Oriented Role Analysis Method (OORAM)

OORAM [35], brought to life by Trygve Reenskaug in 1996, addresses the shortcomings of object orientation and is centered about the concept of role models in which one can capture phenomena as descriptions of object-collaboration patterns. The roles in a role model are archetypical representatives of objects that will occupy corresponding positions in the object system, and all objects that take the same position in such a pattern are said to *play the same role*. Object patterns that enact the roles are *role model instances* (where each object in the pattern is a *role instance*).

As the roles in a role model have an identity and are encapsulated (like objects in basic object orientation), the role models are in fact object oriented models of object structures. Thus, role models are actually abstractions over object structures, and, if used properly, narrow the translation gap between mental models of real world phenomena and software elements (i.e. the object system). This allows for improved separation of concerns by enabling design

of large and complex systems as a synthesis of multiple role models, where each role model represents a separate area of concern. The role models in OORAM can be seen as small, per-concern metamodels or recipes that define how the individual concerns are implemented. Hence OORAM also facilitates the design of reusable components.

OORAM never became as industrially popular as one would expect, but clearly many of its core ideas are present in recently emerging design paradigms and in state-of-the-art research.

3.1.2 Unified Modeling Language (UML)

During the object revolution, several techniques for *object modeling* emerged from different isolated communities around the computing world. Thanks to organizations like The Object Management Group (OMG) who early recognized and pushed forward the need for standardization, the best practices of the different communities have been melted together to form the Unified Modeling Language (UML) [18-20, 36], which now is generally appraised as the de facto standard modeling language in the field of software engineering.

UML, in its current form, can be characterized as a general-purpose modeling language, and can be used to specify, visualize, construct, and document software systems as well as model business processes, represent organizational structures and more. UML is a great aid for concern separation during software design as it enables the creation of abstract system models from different views (e.g. structural, behavioral, architectural), all using a standardized graphical notation. Hence it becomes easier to design and reason about large and complex systems as humans are better at working with graphical models than with code. With a unified language for modeling, the evolution of model driven engineering techniques has accelerated rapidly the recent years. Model driven engineering is described in chapter 3.4.

To make it possible to capture rules and constraints that are not expressible by diagrammatic notation alone, the Object Constraint Language (OCL) [37] has come to life as a supplement to UML. OCL is a well-defined, textual language that enables definition of constraint expressions free of both natural-language ambiguities and the difficulties of complex mathematics. Together, UML and OCL form the fundamental, enabling technologies for model driven engineering (chapter 3.4).

3.1.3 Design Patterns

Comparing and recording the design decisions taken by many software developers to solve a particular design problem will over time yield a general repeatable solution to it. These general repeatable solutions are termed *design patterns* [38] and can be viewed as templates for avoiding poor system design. Design patterns contribute to improved separation of concerns by providing software developers and architects with pre-existing design solutions that can be applied with low risk of ending up with intangible systems.

3.2 AGENT-BASED DESIGN

Agent-based software development [39] is centered on the notion of agents which may be viewed as active, more intelligent objects or entities that are not specified directly in terms of a direct input-to-output mapping (i.e. in terms of what to do) but in more sophisticated terms like knowledge, beliefs, goals, responsibilities, roles, actions, reactions, etc. (i.e. in terms of how to decide what to do). An agent-based system is a system of multiple interacting agents that autonomously pursue their own agendas, alone or in cooperation, based on inherent knowledge and continuous sensing of their environment. Such systems may be realized on top of distributed object orientation platforms, and several languages and frameworks for constructing agent-based systems exist.

Agent terminology and principles lend themselves more naturally to translate requirements into software design, especially for construction of systems that are distributed over networks and autonomous, reactive, complex and intelligent in their nature. Hence, we can say that an agent-based design approach contributes to separation of concerns by enabling higher-level, goal-oriented design of systems.

Agent-based software development does not replace object orientation and is simply a natural evolution of object orientation principles to meet the needs of social, interacting software with the previously mentioned attributes. The agent approach is rather complementary to object orientation as in most agent-based systems there will be plain, passive objects present as well, that are controlled, accessed and modified by agents.

3.3 SERVICE- AND COMPONENT-ORIENTATION

3.3.1 Service Orientation

Service orientation [40] is centered on the notions of services as network endpoints and connections among them, and comprises a set of common design principles that realize services in ways supporting strategic and architectural goals associated with *Service Oriented Architecture* and *Service Oriented Computing*.

The key principle in service orientation is to build independent services that have well-defined interfaces through which they can be called to perform their tasks in a standardized manner, without any foreknowledge of the client application and without the client having knowledge of how the service actually performs its tasks. This facilitates loosely coupled and interoperable, black-box software elements (e.g. services) that can be hierarchically composed and distributed across organizations and geographical areas.

3.3.2 Component Oriented Design

Many of the design principles of Service Orientation are rooted in, or similar to, a more established design paradigm known as Component Orientation [41], which focuses on constructing software systems as (hierarchical) compositions of reusable building blocks (of software) called components. What exactly constitutes a component is a question to which the answer varies in different communities; however a widely cited definition is that of Szyperski [41]:

> *"A software component is a binary unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*
>
> - Clemens Szyperski, [41]

Components are meant to be delivered and deployed independently in package as binary code along with their required resources, such as images and libraries, and eventually statically connected with other components. This is in contrast to service orientation, where services are connected or composed in a more dynamic fashion.

Service- and component orientation are natural evolutions of object orientation to meet the needs of more componentized software design and assembly. Like agent-based design, service- and component orientation contribute to separation of concerns by providing higher-levels of abstraction than the concept of object, and thus facilitate a higher-level, goal-oriented design (i.e. more natural translation of requirements into software design) in larger, distributed systems.

3.4 MODEL-DRIVEN ENGINEERING

Seeking automation, or semi-automation, of technical processes lies at the heart of engineering as a discipline. Successful automation of repeated work can boost efficiency by reducing the time it takes and make it significantly less error-prone and less complex to manage. While it is clear to everybody that main purposes of computers and software systems have been to provide such automation or semi-automation to business, industry and individuals, people often forget to think of the vast levels of automation that lie behind the ever-growing ubiquity of computers and software. Continuously automating translation from increasingly higher levels of a human-understandable language down to hardware-interpretable instructions is what has enabled this ubiquity and its growth.

The paradigms discussed so far have made it easier to translate real-world phenomena into software systems. With UML and OCL and other similar standards, software development has been mechanized further by enabling software developers to visualize, document and exchange system designs in better, more effective ways than earlier. The next brick in reducing the

translation gap between mental models of real-world phenomena and software systems would naturally be a development paradigm that can provide more systematic use of models throughout software development lifecycles and provide mechanisms for automating translation of models into functional systems (i.e. compilable/interpretable application code and other software artifacts). Model-Driven Engineering [9-13] is the realization of this long envisioned paradigm.

In the context of software, Model-Driven Engineering is an umbrella term for multiple branches of model-driven software development like Model-Driven Architecture (MDA) [11, 12] and Software Factories [10]. Differences set aside; the key principle in these techniques is to shift software development from a code-centric process to a model-centric process where Platform Independent Models (PIMs) at high abstraction levels are developed using graphical modeling tools, and, in terms of abstraction, horizontally and vertically transformed into other models, and finally at some feasible time to Platform Specific Models (PSMs) which may be program code or other software artifacts. The transformation knowledge may be supplied by software developers or domain experts, or both in cooperation, and can be captured in transformation languages and tools so the transformations can be repeated. Model-driven engineering also allows for performing reverse transformations in order to improve reverse engineering of existing systems and software artifacts developed in traditional code-centric processes [11, 12].

The emergence and mainstream embracement of model-driven software development techniques represents a paradigm shift in software development. Separation-of-concerns effectiveness is improved several levels by being able to work at the high abstraction levels the graphical models can provide, and automation helps in focusing on a more step-wise design of systems where error-prone, time-consuming tasks are automated at the developers' chosen level of comfort. Model-driven engineering also benefits separation of concerns through separation of who designs what in a system; with graphical modeling tools, domain experts need no longer be software developers or IT-experts, domain experts can hence focus on modeling the systems domain, and contribute with domain knowledge which software developers can further utilize to produce transformations to more software-technical models and finally software artifacts.

3.5 ASPECT-ORIENTATION

One great limitation of the paradigms mentioned so far is that they all suffer from a limitation known as the *tyranny of the dominant decomposition*: they only allow for modularization of a system in one way at a time and any concerns that do not align with that modularization dimension end up scattered across and tangled within several modules. Such concerns are called *crosscutting concerns* and as the name states cannot be isolated or encapsulated into a single module/unit of a system as designed with the previously presented design paradigms. Examples of crosscutting concerns are

security mechanisms, logging, quality of service management, transaction control, etc. These are all features for which the program code will be scattered across the various modules of an application (i.e. various objects in an object oriented system) and tangled with the code of other features.

A paradigm that is capable of letting software developers consider such crosscutting concerns in isolation and somehow encapsulate and evolve them separately from the other features of a system has long been desired the software industry. The solution, *Aspect Orientation* [14], addresses this desire and attempts to conquer the problems imposed by crosscutting concerns.

Aspect orientation was incepted in the mid-1990s in the shape of Aspect Oriented Programming (AOP) at Xerox PARC, and has since then been developed further by a global community of academia and industrial research institutes. A set of different AOP languages and aspect oriented development techniques have evolved over the recent years and aspect oriented software development (AOSD) is currently perhaps the hottest of all state-of-the-art development paradigms as it has just started to mature and gain the attention of mainstream software development [14, 42].

In aspect orientation, crosscutting concerns are handled by providing language mechanisms, development strategies and frameworks that allow software developers to specify the crosscutting concerns in isolation as so-called aspects, and define advice and join points that express how and where in the system the aspects should be woven in. Aspects can thus be viewed as a new abstraction mechanism that allows for capturing of crosscutting concerns in a localized manner. The key principle in aspect oriented software development is to allow software developers to design a system as a base system (concerns that are perfectly modularized using traditional formalisms) and a separate set of aspects (concerns that crosscut with respect to the modularization formalism chosen for the base system). Advanced mechanisms in the programming language and/or development method take care of fusing the base system and the aspects to produce the desired system. The fusion, or weaving, is often done during compilation or runtime [14, 42].

Without doubt, aspect oriented software development is a great leap in the direction of a better technique for separation of concerns in software design activities. While the other paradigms presented in this essay have improved separation of concerns effectiveness mainly by allowing abstraction and more natural translation of functional requirements into software design, aspect orientation principles allow for more natural translation of non-functional requirements as well and for construction of more maintainable systems that are easier to evolve. Aspects can also very well be designed in ways that make them easy to reuse across systems. Aspect orientation makes it easier to develop and maintain large and complex systems, and the principles of aspect orientation lend themselves naturally to development of adaptive, dynamically reconfigurable systems. AOSD principles also form the basis for further research on multidimensional separation of concerns.

Nevertheless, aspect oriented software development also has a few negative sides. The most significant of these is that reverse engineering and debugging of systems is more difficult as it requires insight into the weaving mechanisms used for fusing the aspects with the base system, and hence makes developers and testers accustomed to traditional techniques feel uncomfortable with the new ways of reasoning. Also, the lack of proper, mature testing strategies makes it difficult to test the various components in a system before they are woven together to form the desired system. Thus, although aspect oriented software development attempts to reduce side-effects of changes and additions to a system, it introduces new risks in the software development and testing cycles that may need careful consideration.

4 AOMDF

In this chapter we briefly introduce the main concepts of the Aspect-Oriented Model-Driven Framework. The introduction is based on material from [43].

4.1 INTRODUCTION

In chapter 1, we briefly introduced AOMDF (The Aspect-Oriented Model-Driven Framework) as a specific approach for aspect oriented modeling centered about the design-time separation and weaving of crosscutting concerns. AOMDF facilitates weaving of *crosscutting concerns* at the modeling (i.e. platform independent) level.

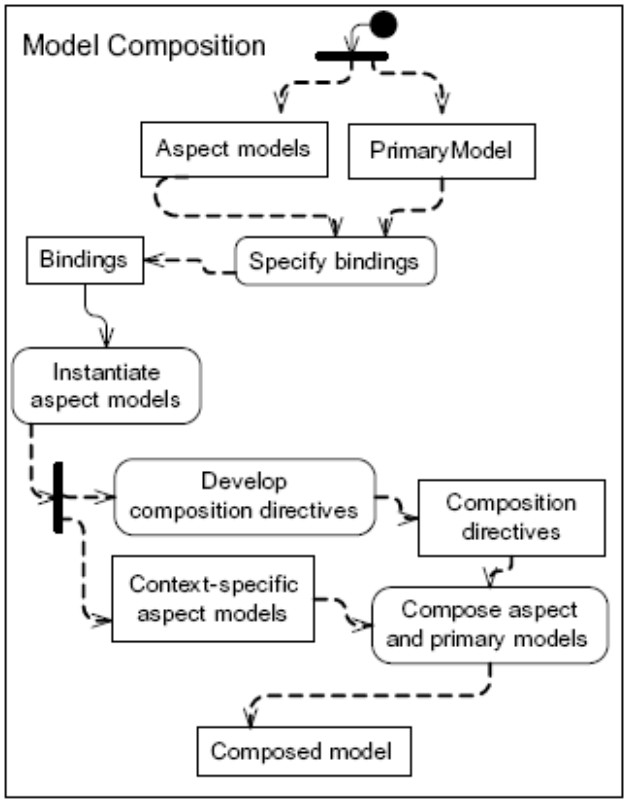

Figure 4-1: Weaving Process in AOMDF (from [43])

AOMDF covers structural and behavioral (visual) modeling of systems in an aspect-oriented fashion through minor suggested extensions to UML [18-20] class- and sequence diagrams (i.e. class- and interaction models). The main contribution of AOMDF is to enable separate, isolated design of the base, core features of a system and its crosscutting features (i.e. functionality that is

spread across various modules of the system and tangled within other functionality). Using AOMDF, system developers may design the crosscutting features as reusable aspect models and the base features as primary models. The primary and aspect models are then woven (composed) together to obtain an integrated view of the complete system. This process is illustrated in .

4.2 INTERACTION MODEL WEAVING IN AOMDF

Weaving sequence models requires one to merge the messages of the primary and aspect sequence models to obtain a corresponding merged sequence of messages in the composed model.

Furthermore, the lifelines in the woven model need to correspond to the lifelines involved in the source sequence models. Thus, the aspect sequence model should be bound to application-specific values before the models can be weaved.

The weaving of message roles in the aspect sequence model with the messages in the primary sequence model requires the specification of how and where the application-specific values of the message templates are to be composed with the primary model messages.

The interaction model weaving technique developed in AOMDF involves tagging the primary interaction model. The *tags* identify the set of primary interaction model elements (e.g., lifelines and messages) to which the aspect interaction model elements needs to be weaved. The technique is analogous to weaving techniques used in aspect oriented programming. The model level tags specify the joint point where a corresponding advice (in the form of aspect sequences) needs to be introduced, and also define the type of weaving that needs to occur between the primary interaction model and aspect interaction model. The interaction model composition occurs under the assumption that the default message flow is described in the primary interaction model. The aspect interaction models can be weaved with the primary interaction model in two ways:

1. By inserting behavior represented by a message or a sequence of messages in the aspect model at a particular point in the primary model

2. By replacing a message or a sequence of messages in the primary model with a sequence of messages in the aspect model.

Two types of tags are applied on the primary interaction model to specify how the aspect sequence model needs to be composed: *simpleAspect* tags and *compositeAspect* tags. The *simpleAspect* tags are associated with a single lifeline and a single point in the primary model while *compositeAspect* tags are associated with multiple lifelines and multiple messages.

We describe these two types of weaving in more detail in chapter 5.3, together with the currently proposed concrete syntax and examples. A more formally expressed description of these two weaving types is presented in chapter 10.1.1.

PART II

TECHNOLOGY FOUNDATION

---

5 Interaction Models in UML2 and AOMDF

In this chapter we provide a basic overview of UML2, UML2 Interaction models and interactions in AOMDF. A vast amount of literature on UML2 is available, so we keep our overview brief. For a more thorough and complete introduction on the usage of UML2 and sequence diagrams we suggest that the reader refers to [36]. Readers interested in the abstract- and concrete syntax, and other underlying technical and historical details, should refer to the UML specifications [18-20]. Most of the material on UML2 in this chapter is based on [18]. The introduction to the existing proposals of the concrete syntax for AOMDF interaction models is based on [4, 5].

5.1 UML2 OVERVIEW

In chapter 3 (section 3.1.2), we introduced The Unified Modeling Language (UML) as the de facto standard modeling language used for logical analysis and design of software systems. UML2 is the second major version of UML, with updates and refinements to both its abstract and concrete syntax based on feedback from industry usage and research on software engineering.

UML2 concepts are divided into six groups of concepts, as follows:

1. **Static structure:** Concepts used for defining the universe of discourse of a system – i.e. the entities constituting the system, their internal properties and relationships to each other. The main diagram type related to this group is the well-known *class-diagram*.

2. **Design constructs:** Concepts used for defining the internal design and ***logical*** architecture of a system, and for describing ***which*** structural entities need to cooperate in order to deliver some functionality. The main diagram types associated with this group are *composite structure diagrams*, *component diagrams*, *and collaboration diagrams*.

3. **Deployment constructs:** Concepts used for defining the run-time implementation, i.e. ***physical*** architecture, of a system. The diagram type associated with this group is *deployment diagrams*.

4. **Dynamic behavior:** Concepts for defining the lifecycles of a system's structural entities, and for describing ***how*** these interact

to deliver some functionality. The diagram types associated with this group are *sequence diagrams*, *state machine diagrams*, *activity diagrams*. *Use case diagrams* supporting use case descriptions also fall into this group.

5. **Model organization:** Concepts used for organizing models into coherent, hierarchical pieces. The diagram type related to this group is *package diagrams*.

6. **Profiles:** Concepts enabling limited, domain specific extensions to UML for ordinary needs, such as including extra, domain specific information into models in a formal way. The extensions are made by creating *profiles* using the notion of *stereotypes* and *tags*. Only lightweight extensions that do not require changes to the abstract syntax can be made using these concepts.

Like we described in chapter 3 (section 3.1.2), UML2 is also supplemented by the Object Constraint Language (OCL 2.0) [37], a language for expressing constraints in UML2 models in a formal fashion.

AOMDF provides support for aspect oriented structure and behavior modeling by extending some of the concepts found in the *static structure* and *dynamic behavior* areas of the UML2 language. The work in this thesis is related to AOMDF's extension of UML2 Inteactions (from the *dynamic behavior* area of UML2).

5.2 INTERACTION MODELS IN UML2

UML2 enables modeling of dynamic behavior inside a system from different perspectives. One of these is the perspective of an object's lifecycle in the system, which can be described in the form of a state machine diagram (using state machine model elements). A second perspective is that of a set of objects realizing some required functionality by systematically interacting with each other using messages (operation calls or signals). This perspective can be described using the Interaction concepts of UML2 upon which the notation used in sequence diagrams is built. AOMDF interaction model weaving is based on UML2 sequence diagrams.

In Figure 5-1, we show a trivial sequence diagram and highlight the essential concepts. The objects taking part in the collaborative effort to supply the required functionality are shown as lifelines representing roles (unpopulated or populated by a known object). The lifeline is made up by a head and a dotted line showing the lifetime of the object. Messages are drawn between the lifelines of the object to show the flow of control, and the vertical dimension represents the passage of time. A double fully drawn line representing an execution specification is used to describe activity in an object, i.e. operation execution and the time the object must wait for execution of nested operations calls. The endpoints of the message arrows represent occurrence specifications

that capture the concept of something occurring, i.e. the trigging of an event (e.g. sending or receiving an operation call).

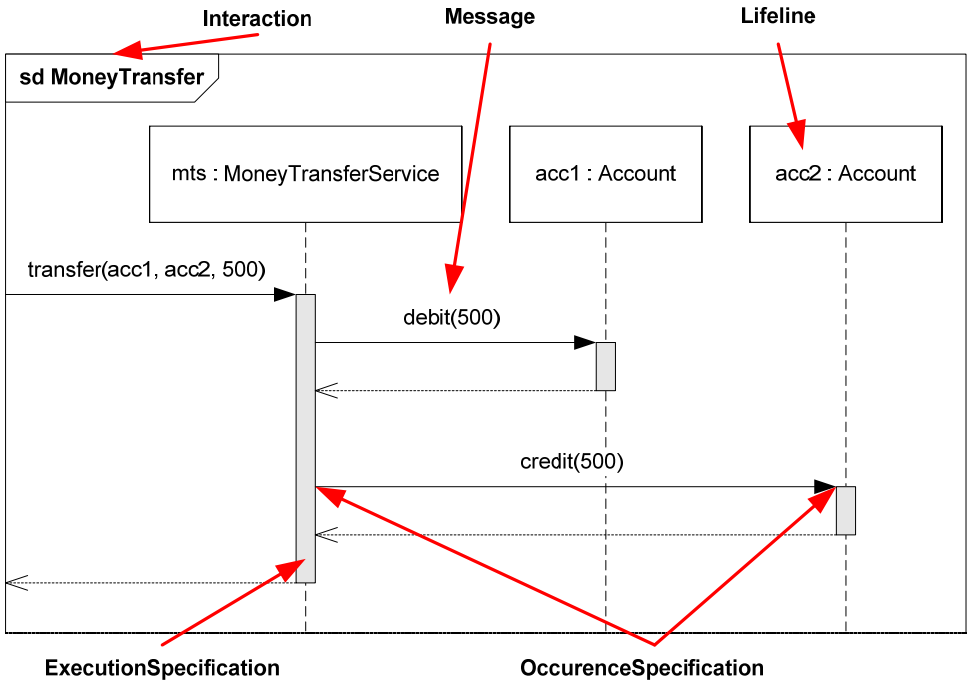

Figure 5-1: A simple UML2 sequence diagram

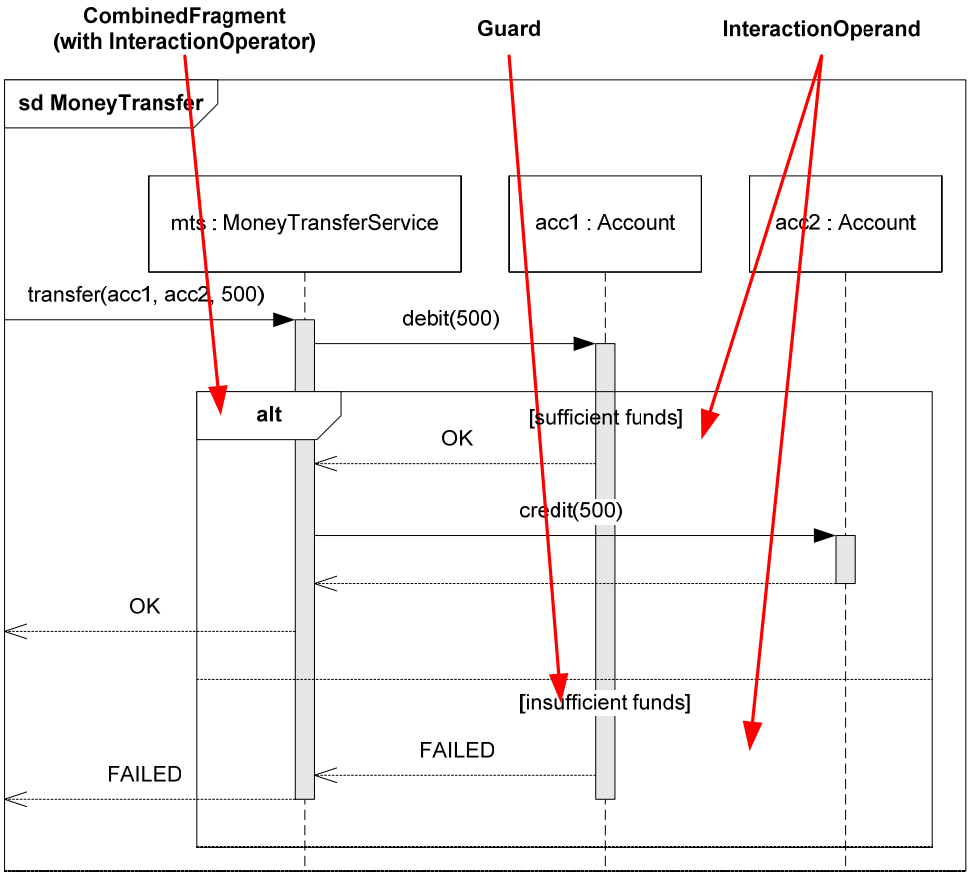

Figure 5-2: UML2 sequence diagram with combined fragment

In Figure 5-2, we show a slightly refined version of the sequence diagram in Figure 5-1. The diagram is refined with a combined fragment in order describe a more complex message flow. The combined fragment is defined with an interaction operator – in this case the alt-operator which can be compared to a branching construct in programming. Depending on the interaction operator a combined fragment may contain one or more interaction operands.

We do not indulge further into describing the remainder of the UML2 interaction concepts as this is widely available in referenced literature and in the UML2 specification [19, 20]. (The concepts shown in our figures should be sufficient for understanding the subsequent chapters).

5.3 INTERACTION MODELS IN AOMDF

In chapter 4 we introduced AOMDF – The Aspect Oriented Model Driven Framework – as an aspect oriented modeling approach that allows pervasive system features to be designed in isolation and adapted into core, pervasive-feature-free models of various systems. We described there the modeling strategy of AOMDF, and briefly mentioned interaction model weaving. Having introduced UML2 interactions, we now introduce the notation (i.e. concrete syntax) proposals of [4] for describing aspect interaction models and adding weaving instructions in primary interaction models. Primary interaction models are ordinary UML2 interactions.

AOMDF defines two kinds of weaving of interaction models; the *simpleAspect* weaving and the *compositeAspect* weaving. The first of these is a mere adaptation of the message sequence from an aspect interaction into a certain point in a primary interaction. The second kind is a more complex adaptation in which the message sequence of the aspect interaction is woven into the primary interaction's message sequence according to advice in the aspect interaction. We respectively describe the two weaving kinds and the previously proposed notation associated with each.

5.3.1 Simple Interaction Aspects

The *simpleAspect* weaving in AOMDF is based on the idea of simple interaction aspects that can capture simple pervasive features like basic logging. An example of a simple aspect interaction is show in Figure 5-3. As observable, the only difference between the aspect interaction and ordinary UML2 interactions is that the first one contains a *lifeline-role* and an *argument-role* that must be populated respectively by a lifeline and some value when adapted into a primary model. The lifeline-role is shown with the sign "|" in its head, which is similar to the notation for templates in the UML-based role- and pattern modeling language RBML[44-46].

Figure 5-4 shows a primary interaction into which weaving instructions have been tagged as stereotyped messages. It is intuitively understood that the source and target lifeline of each such stereotyped message is the same, and

hence that primary lifeline is the one that will populate the lifeline-role upon weaving. Figure 5-5 shows the final woven model. Notice the weaving of the two lifelines and argument binding in addition to the woven message sequence.

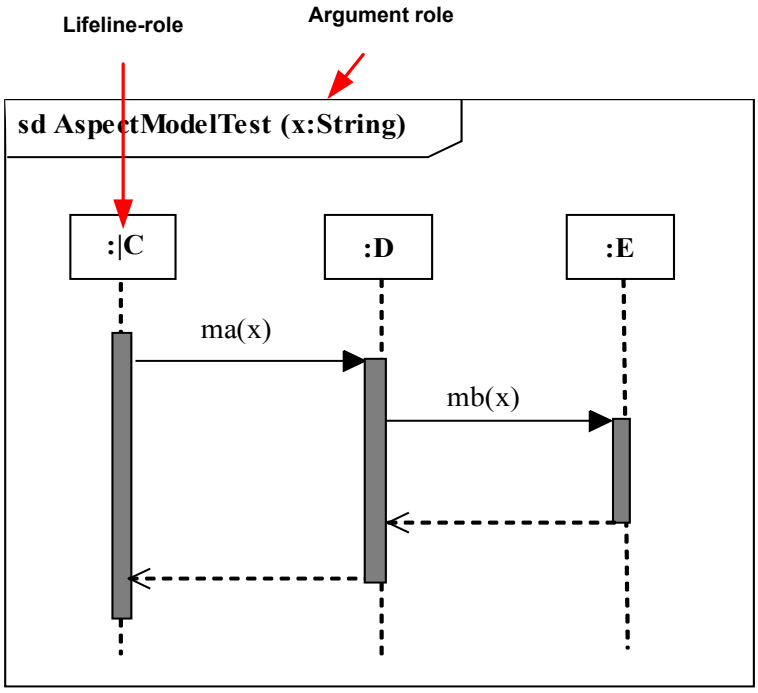

Figure 5-3: Example of AOMDF simpleAspect interaction model

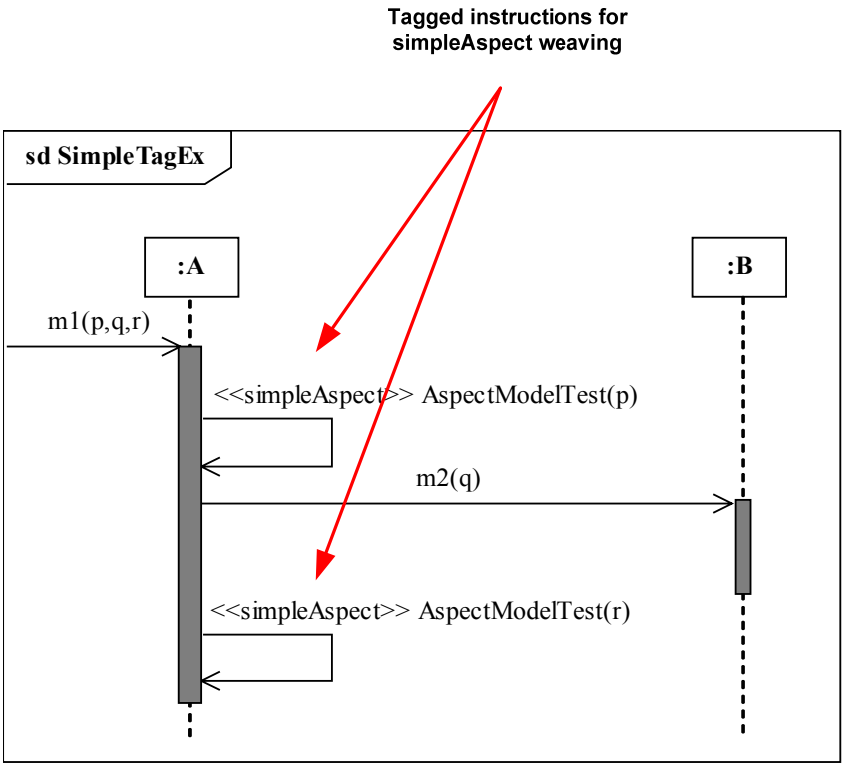

Figure 5-4: Requiring weaving of a simpleAspect in a primary model

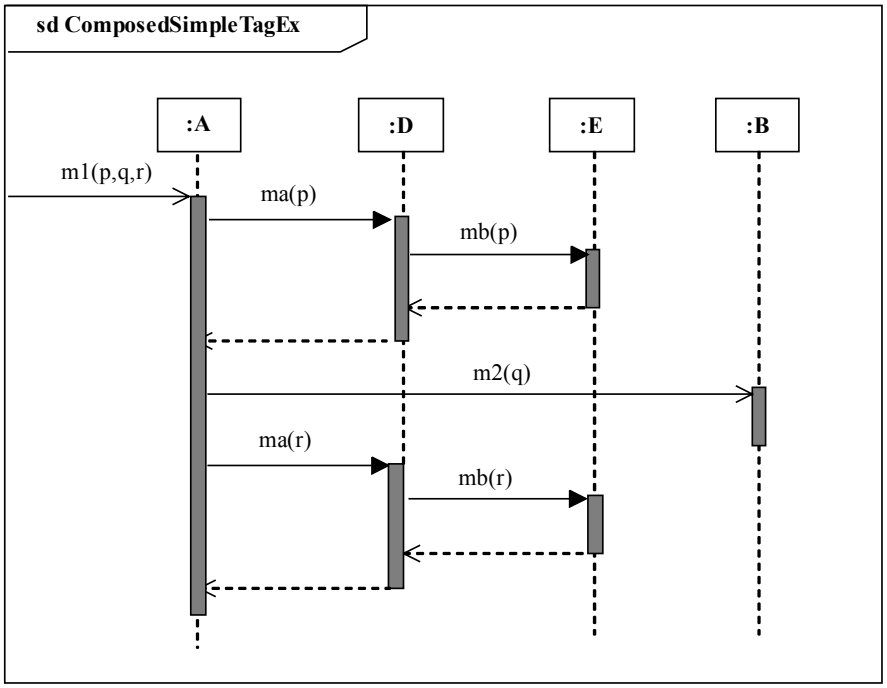

Figure 5-5: Result of weaving SimpleTagEx and AspectModelTest.

5.3.2 Composite Interaction Aspects

The *compositeAspect* weaving in AOMDF is based on the idea of composite interaction aspects that can capture more sophisticated and super-pervasive aspects like transaction management, QoS monitoring and similar that are heavily tangled within core system functionality. Such super-pervasive concerns can not be modularized into simple pieces to be woven in at desired points in primary models. They need to be instrumented carefully into different segments of a primary model (and not just at one single point) upon each adaptation. Analogous to the programming level concept of advice in AOP, this is accomplished in AOMDF by grouping the messages in the aspect interaction according to how they should be woven into the primary models during adaptation. In Figure 5-6, we show how a combined-fragment-like notation is used to represent the message advice. (The various operators of these fragments is explained later when we develop the abstract syntax for these).

Figure 5-7 shows an example of a composite aspect interaction. Notice that one of the advice messages is a message-role, which upon population during weaving will refine messages in the primary interaction. Figure 5-8 shows how weaving instructions for *compositeAspect* weaving is tagged into primary interactions. Again a combined-fragments-like notation is utilized, embracing a message subsequence over which the advice in the aspect interaction is to be adapted.

Finally, Figure 5-9 shows the resulting interaction obtained by weaving the aspect interaction in Figure 5-7 into the primary interaction in Figure 5-8.

Notice how the excluded message *m3* is left untouched by the weaving process.

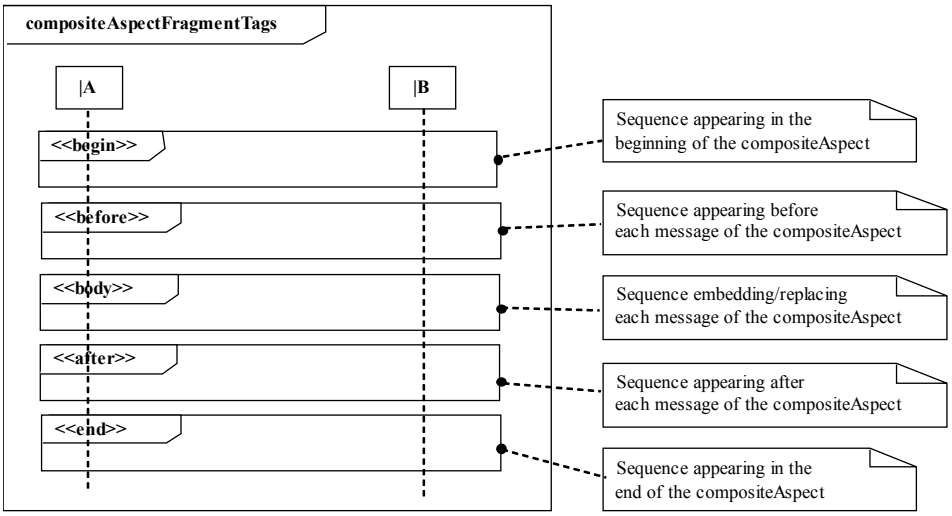

Figure 5-6: Use of combined-fragment-like notation for specifying aspect advice.

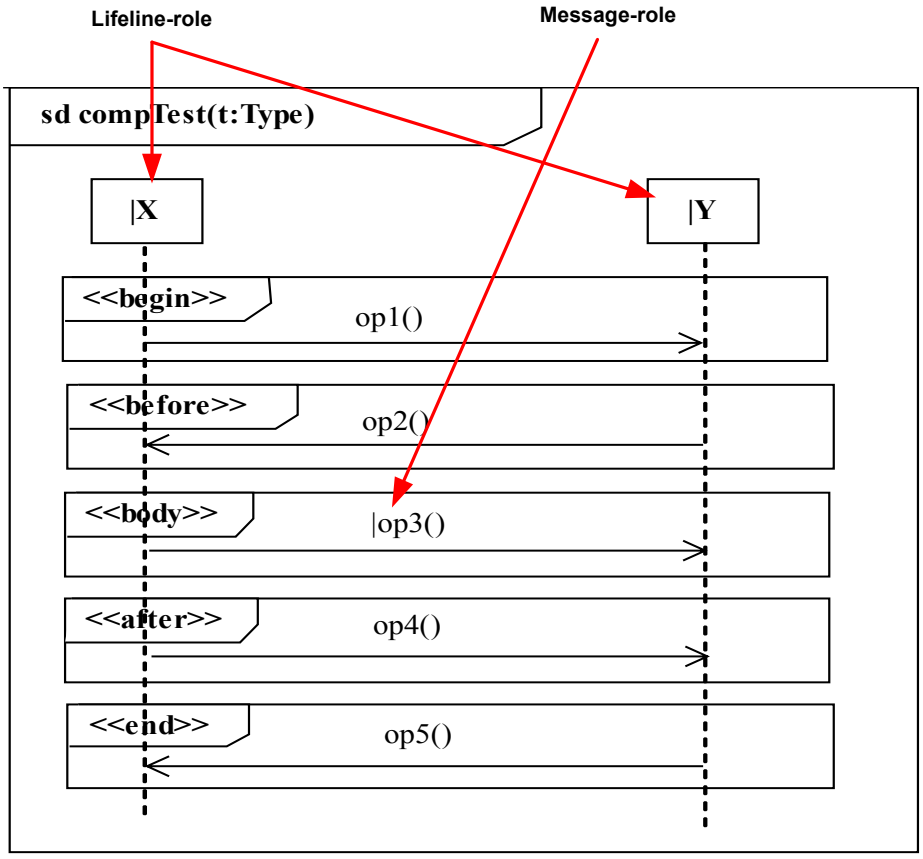

Figure 5-7: Example of AOMDF compositeAspect interaction model

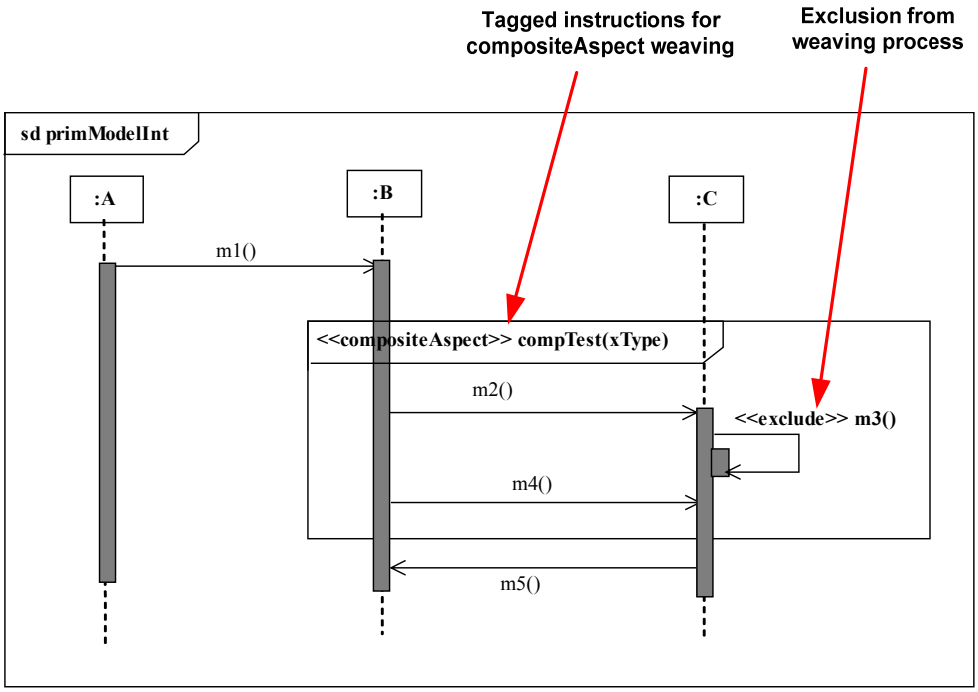

Figure 5-8: Requiring weaving of a compositeAspect in a primary model

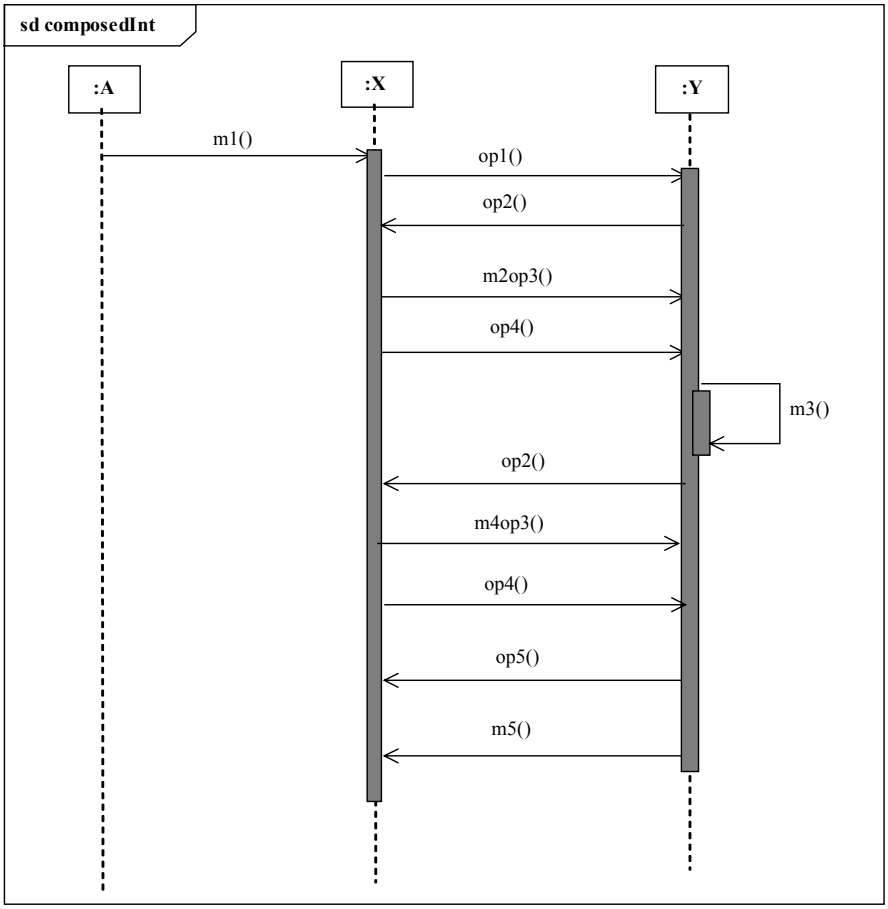

Figure 5-9: Result of weaving primModelInt and compTest

6 Metamodelling and EMF

In this chapter we briefly introduce the concept of metamodeling, metamodel levels and formalisms.

6.1 METAMODELLING

A metamodel is a model of a modeling language, i.e. a more abstract description of the language. The metamodel of a language precisely defines the concepts, grammatical relationships, and other rules needed to construct well-formed, semantic models in that language. A metamodel is itself also a model, which in turn is defined by a meta-metamodel [27].

Metamodels can be thought of as analogous to the *BNF* (or *EBNF*) grammar of a programming language. Data models, schemas and other meta-descriptions of some domain specific vocabulary are all metamodels for that particular vocabulary. In short, like models are abstractions of phenomena in the real world, metamodels are higher-level abstractions defining the properties of models [11, 12, 27].

Metamodelling (the activity of engineering metamodels and languages) has become widespread with the emergence of model driven engineering. The common way of creating domain specific modeling languages is by seeking out proper design abstractions from the problem domain and capturing these in a metamodel. Very often this is done by extending an existing modeling language, or a tailored subset of an existing language. In Model Driven Architecture (MDA) [11, 12] languages are unambiguously defined as metamodels.

6.2 METAMODEL ARCHITECTURE

A model *(M)* must always conform to a metamodel *(MM)*. Since the metamodel (*MM)* is itself a model, it conforms to a metamodel *(MMM)*. In terms of classification, "*conforms to*" means the existence of an "*instance of*"-relationship, i.e. all concepts in the model *M* are *instances of* some concept in metamodel *MM*. This could continue to infinite levels, with infinitely many models at all levels. However one seeks to unify at some meta-level so that, ideally, all models either directly or indirectly conform to the same metamodel. This way translation between languages and semantic interchange of data becomes possible as the common metamodel provides the bridge between concepts of the languages conforming to it.

The Object Management Group (OMG) defines a four-layer, closed metamodeling architecture to classify models into *modeling levels* (or "*meta-levels*") [11, 12]. The layers are named *M0*, *M1*, *M2* and *M3*. The sole purpose of these layers is to provide a common set of reference frames for discussing models and metamodels. An overview of the layers is shown in Figure 6-1, and the layers are described below.

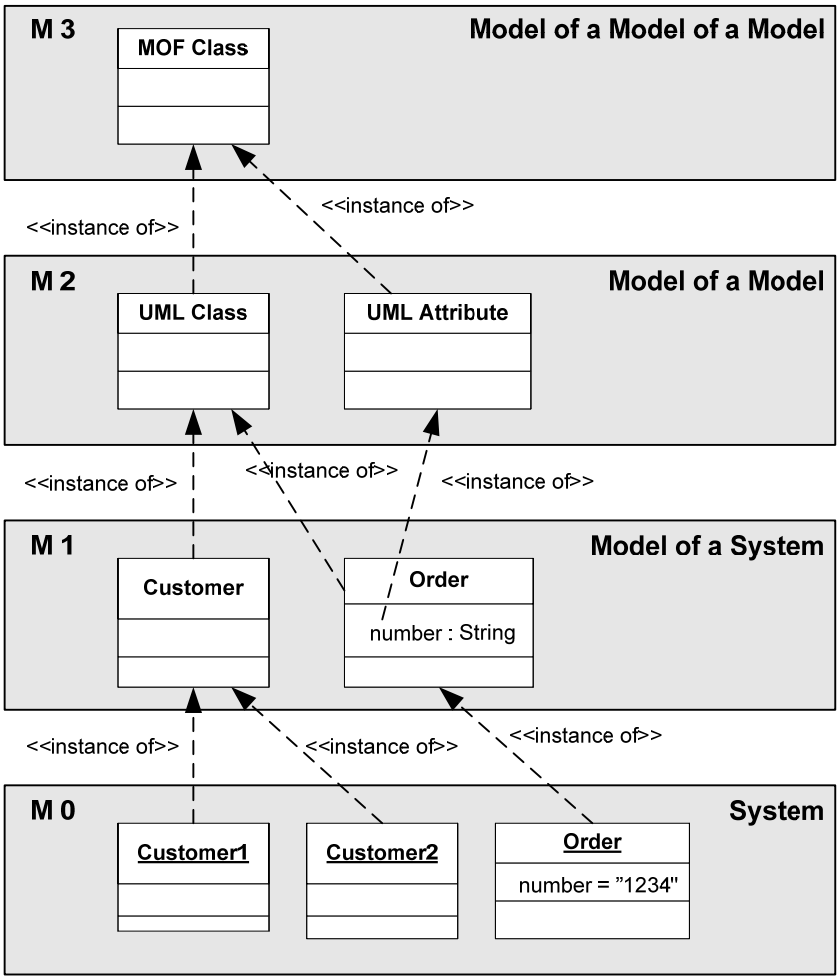

Figure 6-1: Overview of the four modeling layers defined by OMG

The lower-most layer, M0 (system), is where the representations of real world items/data (i.e. objects in an object oriented system) exist. The layer M1 (model of a system) contains models specifying the instances at M0. The concepts at M1 *classify* the instances at M0. As an example, a UML model containing a system design is considered to be at layer M1, while the objects in the deployed, running system will exist at M0.

Concepts at the M1 level are in turn specified by concepts at the M2 level (model-of-a-model). In the case of UML, the UML metamodel exists at level M2 (while a UML model exists at level M1). Similarly, the concepts in the UML metamodel at level M2 are specified by higher-level concepts at M3 (model-of-a-model-of-a-model).

At the M3 layer, all elements are required to be instances of concepts of the M3 layer itself. In other words, models at M3 are, circularly, their own metamodels (i.e. self-descriptive languages). Models on the M3-level are commonly referred to as meta-metamodels.

6.3 METAMODEL FORMALISMS

Metamodels are modeled in modeling languages made for that purpose, i.e. domain specific languages made for defining languages. The meta-meta-models (at level M3 in the four-layer architecture) are such metamodeling languages.

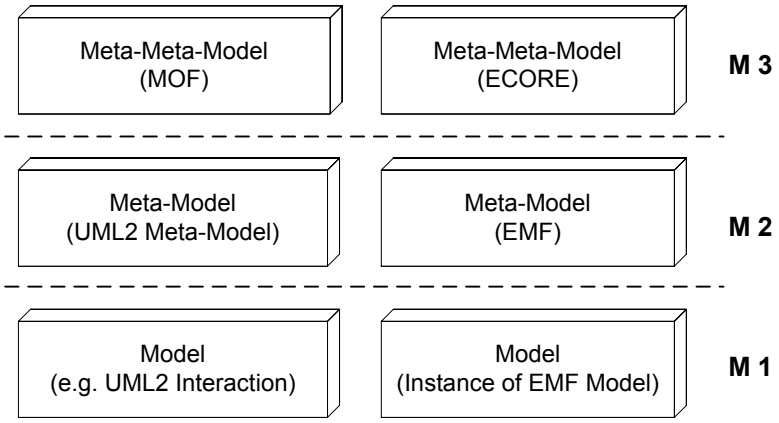

Figure 6-2: Meta-meta-model formalisms

6.3.1 Meta Object Facility (MOF)

The Object Management Groups' standard meta-meta-model is the MOF (Meta Object Facility) [47]. MOF is self-descriptive, and the most commonly known M2-level model based on MOF is the UML metamodel[2]. In Model Driven Architecture all languages and tools are meant to be based on MOF. Other branches of model driven engineering utilize other substitutes or variants of MOF.

6.3.2 Eclipse Modeling Framework (EMF) and ECORE

The Eclipse Modeling Framework [48] is an extension to Eclipse providing a tool- and technology foundation for working with models and developing modeling languages and tools. EMF implements a tailored, optimized subset of the MOF specification. The core, MOF-like, metamodel in EMF is called ECORE.

[2] MOF actually originates from early work on specification of UML

In order to use UML within EMF, an ECORE based implementation of the UML metamodel has been created.

6.3.3 Essential MOF (EMOF)

The new proposal for MOF 2.0, introduces an EMF-like subset of MOF as a separate variant. The EMOF and ECORE meta-meta-models only have minor differences and EMF is capable of handling models that conform to EMOF.

In our work in this thesis, we utilize an EMF-based tool for developing our metamodel, and an EMOF based language for implementing our model weaver. These however seamlessly work together on the EMF platform. The canonical representation of models in these tools is based on XML Metadata Interchange (XMI) [49].

7 Model Transformation and Kermeta

This chapter briefly introduces model transformations, transformation definition tools and languages, and Kermeta.

7.1 MODEL TRANSFORMATIONS

A central concept in model driven engineering is the automation of tedious, repeatable, error-prone tasks in a software development process. A software development process can be viewed as a chain of transformations that translate information from abstractly expressed requirements to deployed, executable systems. While the need for human intervention and craftsmanship is inevitable in parts of this chain, other parts may very well be fully automated. Model transformations are the building blocks needed to facilitate such automation.

A precise definition of model transformation is given in [12]:

> *"A transformation is the automatic generation of a target model from a source model, according to a transformation definition."*
>
> *"A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language."*
>
> *"A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language."*
>
> - Kleppe, et al [12]

A model transformation from some source model *Ma* to a target model *Mb* is illustrated in Figure 7-1. Each of the models conforms to separate metamodels, and these in turn conform to a meta-metamodel. The transformation conforms to a mapping containing the transformation definition between the concepts of the respective metamodels. The mapping is in turn expressed in a transformation language which also conforms to the meta-metamodel.

Weaving of models can be considered as a complex model transformation that takes two or more source models as input and combines them into a single target model.

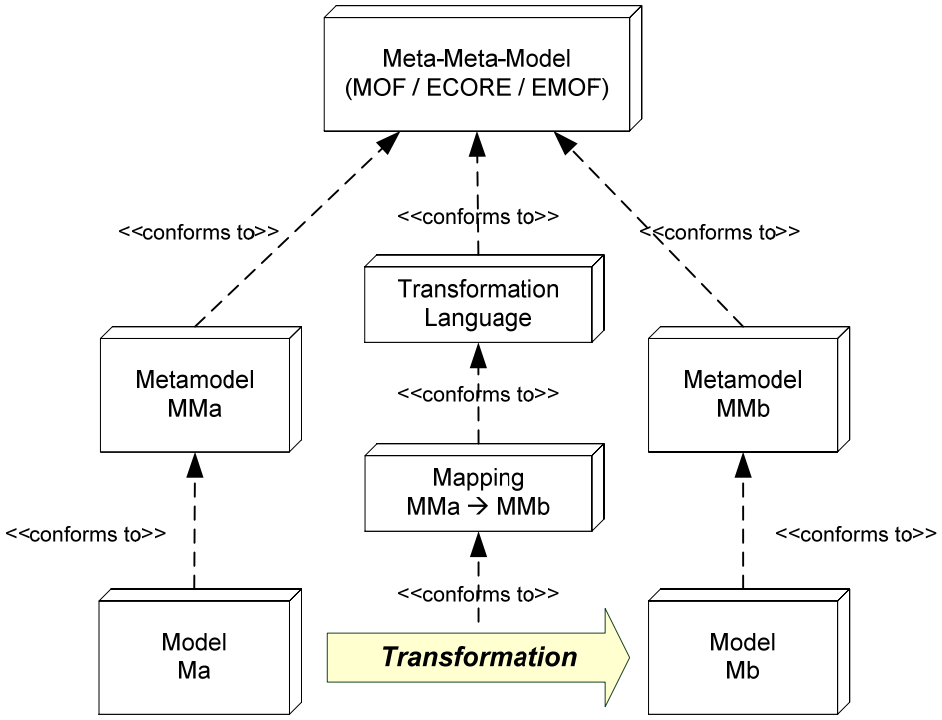

Figure 7-1: Model transformation

7.2 MODEL TRANSFORMATION LANGUAGES

Several model transformation languages have emerged along with model driven engineering. Some of them specifically target the definition of model-to-model transformations, like The Atlas Transformation Language (ATL) [29]. Other are targeted for the definition of model-to-text transformations (e.g. code or documentation generators), like MOFScript [50]. Most of the transformation languages are simple, rule-based languages and their modularization constructs do not scale as the transformations get complex or large in size.

7.3 KERMETA

Kermeta [32] is an open source metamodelling and metaprogramming language developed by the Triskell Team at IRISA. The vision of Kermeta is to provide a common basis for implementing metadata languages, action languages, constraint languages, complex transformations and model weavers. Kermeta can be used to both define and alter the structure and behavior of a user-designed metamodel.

The KerMeta metamodel is divided into two packages: structure and behavior. The structure corresponds to the OMG metamodeling language Essential Meta-Object Facility (EMOF) [47]. The behavior corresponds to actions. Kermeta is compatible with EMF and can directly work on any ECORE-based models.

KerMeta's action language includes object-oriented features and model-specific features. Some of these model-specific features are used in our research for the implementation of the model composition. For example, association end references have a property called opposite which makes it possible to model the opposite association end reference to which it is associated. Some other features like object containment have been included. In addition to this, KerMeta implements OCL closures such as *each*, *collect*, and *select*. Recent releases of Kermeta also introduce aspect-orientation-like modularization features providing better separation of concerns in complex model tranformations.

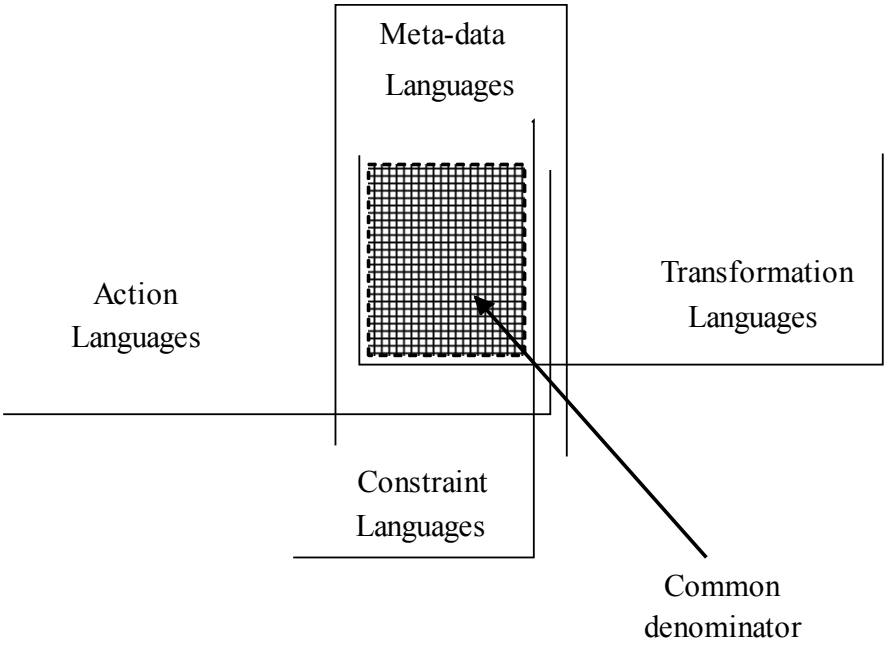

Figure 7-2: Positioning of Kermeta (from [24])

Based on its obvious strengths beyond those of traditional model transformation languages, we have selected Kermeta for the implementation of the model weaver developed in this thesis. We detail our reasons for this choice in chapter 8.

PART III

CONTRIBUTION

8 Technical Solution Approach

This chapter presents the technical solution approach for the development of the metamodel and the model weaver, respectively. A solution strategy for each is devised based on the identified requirements and expected challenges.

8.1 METAMODEL

As mentioned in chapter 1, section 1.3.2.1, our initial goal is to design the abstract syntax of AOMDF interactions in the form of a metamodel. In terms of the metamodelling layers (described in chapter 6, this metamodel will function as the M2-layer (i.e. model-of-a-model) for AOMDF. Below, we identify the requirements and challenges associated with the metamodel development and outline a solution strategy.

8.1.1 Requirements

Obviously, the main requirement for the metamodel is that it should be fit for its purpose, i.e. function as an abstract syntax for AOMDF interaction models and allow us to realize the proof-of-concept for interaction-model weaving. However, the satisfaction of the above goal does not necessarily imply that the metamodel is of high quality. A high-quality metamodel is desirable to avoid unwanted complexity during development of the model weaver and for easier comprehension of the AOMDF language. Thus, to ensure an acceptable level of quality, we state the following list of requirements which we must seek to fulfill in addition to the main requirement:

3. **Behavior-structure-completeness**[3]**:** Interaction models are used to describe behavior between structural elements. An interaction model can thus, from an external point of view, be considered as incomplete without an associated structural model (e.g. as described in chapter 5, a *Lifeline* in a sequence diagram represents a structural element like *Property*, which in turn may be typed by a *Class*). The metamodel should facilitate the construction of *behavior-structure-complete*[3] models, i.e. models that contain

[3] The term *behavior-structure-complete* is not found to be used by anyone before, however we might as well coin it as it does make sense to say that a model is *behavior-structure-complete* (or, if you like, *structure-behavior-complete*) whenever the model contains both behavioral- and structural descriptions of a certain system or part of a system.

interaction elements alongside the structural participants of the interaction. In other words this means that the metamodel should cover both the structural and behavioral concepts of AOMDF.

4. **Compactness:** The metamodel should not contain any unnecessary concepts, i.e. the number of meta-classes should be kept to a minimum to avoid bloating the AOMDF abstract syntax. This includes reuse by specializing existing concepts whenever possible and extending existing metamodels (for example the UML metamodel).

5. **Well-formedness:** The metamodel should be well-formed, i.e. illegal models should be ruled out by means of well-formedness constraints, preferably stated using OCL.

6. **Extensibility:** The metamodel should be easy to extend in order to allow for extension of AOMDF with other kinds of UML models and diagrams (such as Composite Structures).

7. **Portability:** Although the metamodel will reuse other metamodels (such as the UML metamodel) and conform to the meta-metamodel of the environment we select for implementation, we should seek to design our metamodel in a way that makes it fairly easy to port it to environments based on other meta-metamodels.

8.1.2 Challenges

The attempt to adequately fulfill the requirements listed in the previous section is in itself a great challenge. However, we identify two even bigger challenges that we need to address in our solution strategy.

8.1.2.1 Immature Abstract- and Concrete Syntax Legacy

Previously publicized work on AOMDF proposes a split concrete- and abstract syntax for AOMDF interaction models based on an idea of combining both customization and extension of UML. As presented in chapter 4, the aspect models are suggested to reuse concepts from the UML-based pattern-specification language RBML [44-46], while the tags applicable on primary models are suggested to utilize stereotypes and the UML profile mechanism to customize ordinary UML notation. This split approach may seem reasonable when considered at the theoretical level, however looking towards implementation of our model weaver, we recognize a need for a more unified abstract syntax model.

A single unified metamodel which describes the abstract syntax concepts needed for both primary- and aspect models, is not only beneficial for a simpler and more straight-forward implementation of our model weaver, but also more in line with our requirement of portability and with the teachings of Language-Driven Development preachers like [26, 27, 51], who discourage to lean too much on the profile mechanism.

A comprehensible summary of the advantages and disadvantages of using UML profiles to build domain specific languages was presented in [52]. The key disadvantage listed was:

> *UML profiles only permit a limited amount of customization. It is not possible to introduce new modeling concepts that cannot be expressed by extending existing UML elements. (...)*
>
> - Extract from Table 1 in [52].

Based on all the above arguments, we claim that the current abstract- and concrete syntax of AOMDF is somewhat immature. Hence, we cannot simply derive our new abstract syntax model from the proposed concrete syntax by means of a simple mapping. A slight reconsideration of parts of the concrete syntax will also be necessary during the metamodel development.

8.1.2.2 Navigational Complexity of the UML2 metamodel

Proper reuse and extension of the visual notation of UML2 requires an in-depth understanding of relevant parts of the UML2 abstract syntax. However, the UML2 metamodel specification [19, 20] is claimed to be quite complex to deal with as a consequence of its reliance on the design principles *high cohesion* and *low coupling*. These principles have certainly yielded a functioning internal modularization, but, as a major side-effect, abstractions of closely related concrete syntax elements have been spread far away from each other in the abstract syntax model. This phenomena is described in [53, 54] and we acknowledge that it perplexes the navigation of models.

Figure 8-1, Figure 8-2 and Figure 8-3 illustrate the above described phenomena. The first figure is extracted from the graphical syntax for Interactions in the UML 2.0 specification. The two latter figures are extracted from the abstract syntax model of Interactions. Notice how there is no obvious link between a Lifeline and a Message in the abstract syntax model while in the concrete, graphical syntax, Lifelines and Messages are closely related (as shown in chapter 5.2).

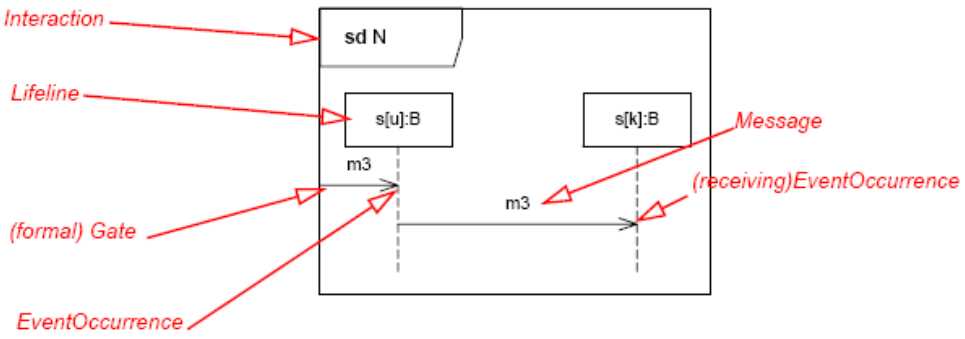

Figure 8-1: Lifeline and Message notation (from UML 2.0 Specification)

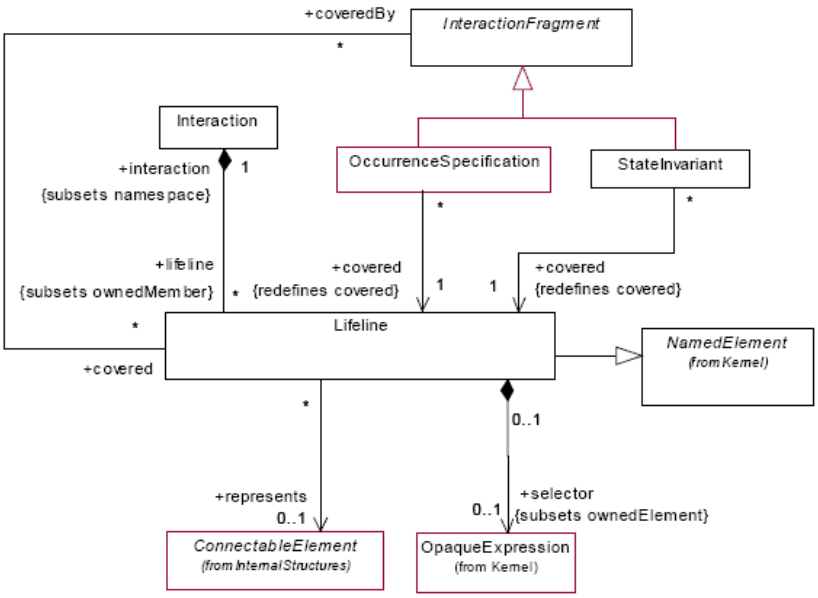

Figure 8-2: Lifelines in abstract syntax (from UML 2.0 Specification)

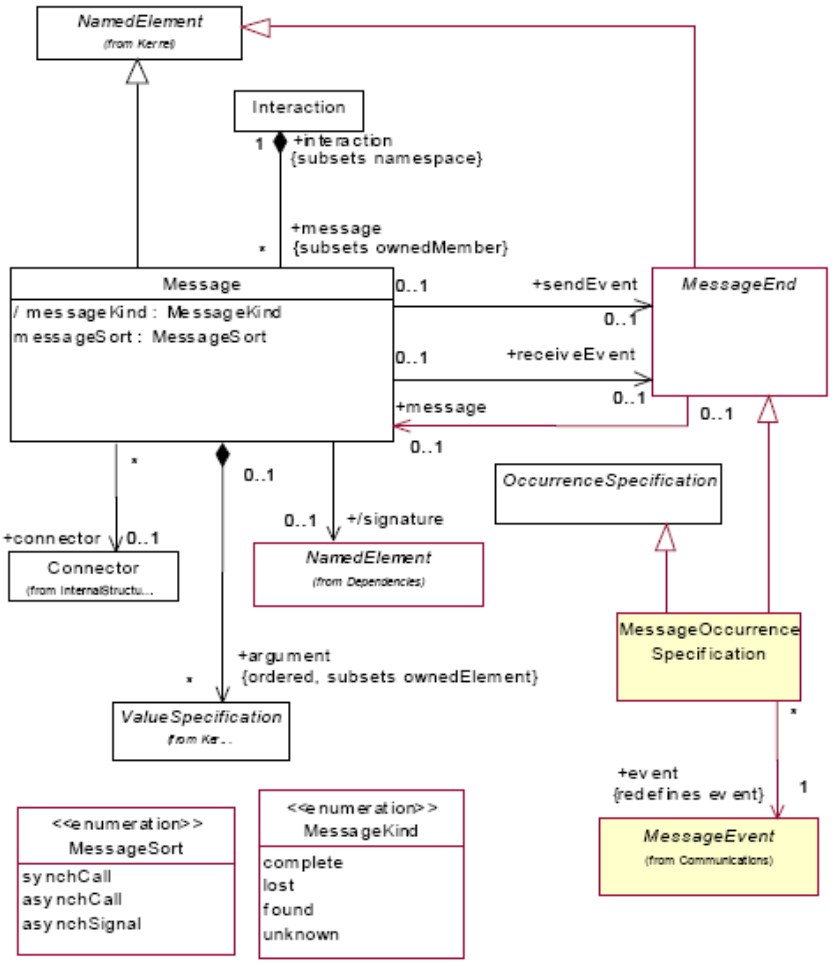

Figure 8-3: Messages in abstract syntax (from UML 2.0 Specification)

8.1.3 Solution Strategy

While devising a solution strategy that targets the identified requirements and challenges, it is essential that we remind ourselves that our main goal is to build a proof-of-concept for interaction model weaving in AOMDF. To keep the main goal within reach, it is crucial that we narrow our scope as much as possible even during the metamodel development.

8.1.3.1 Metamodel Organization and Scoping

As described earlier, the abstract syntax for AOMDF makes direct reuse of a subset of concepts found in UML2 and RBML [44-46], and we may thus divide it into three conceptual layers as illustrated in Figure 8-4.

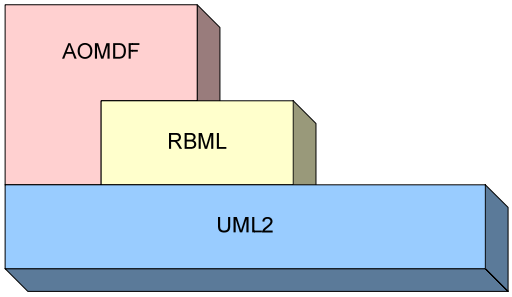

Figure 8-4: AOMDF abstract syntax as an extension of UML2 and RBML

Proper reflection of this layering in our metamodel structure should fulfill the compactness and extensibility requirements. However, we have also required portability across implementation environments (i.e. meta-metamodel conformance portability) and, if possible, we seek to avoid the navigational complexity found in the UML2 metamodel. We believe that these issues can be addressed through rapid construction of simplified metamodels for UML2 and RBML. These simplified metamodels should only contain the subset of UML2- and RBML-concepts that the AOMDF-layer shown in Figure 8-4 depends upon and may introduce simplifications and optimizations that help us to overcome the navigational complexity. If constructed properly, the simplified metamodels can replace the complete UML2- and RBML metamodels – at least for the purpose of our main goal. We illustrate this in Figure 8-5.

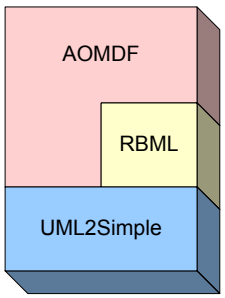

Figure 8-5: Bootstrapping AOMDF with simplified UML2 and RBML

The behavior-structure-completeness requirement is satisfied by capturing both the structural concepts (i.e. the concepts used for class-models) and the

behavioral concepts (i.e. the concepts used for interaction-models) into the metamodel.

8.1.3.2 Concept Identification

The guidelines in [27] suggest identification of abstract syntax concepts by matching language concepts against the following criteria:

- Concepts that have names.
- Concepts that contain other concepts.
- Concepts that record information about relationships between other concepts.
- Concepts that play the role of namespaces for named concepts.
- Concepts that exhibit a type/instance relationship.
- Concepts that are recursively decomposed.
- Concepts that are part of an expression or are associated with expressions.

In our case, the abstract- and concrete syntax legacy should be our primary source of knowledge during concept identification. We should however ensure that our metamodel contains only the core modeling concepts, and avoid the pitfall of "modeling the diagrams", by checking against the criteria above and consider the following questions [27]:

- Does the concept have a meaning, or is it purely for presentation? If the latter, it is concrete syntax (and not abstract syntax).
- Is the concept a derived concept or is it just a view on a collection of more primitive concepts? If the latter, a relationship should be defined between the richer concept and the more primitive concept.

8.1.3.3 Constraints

We ensure the well-formedness requirement by providing a set of constraints expressed in OCL [37] that rule out illegal models. However, we do not make any effort to provide a complete set of constraints.

8.1.3.4 Implementation

Based on our intentions to use Kermeta to implement our model weaver, we naturally implement the metamodel as an Ecore-model in an EMF-based, graphical modeling tool – The TOPCASED Ecore Editor [55]. The metamodel is stored in an ecore-file which can be dynamically loaded by the Kermeta engine.

The resulting metamodel is presented in chapter 9.

8.2 MODEL WEAVER

The second goal of this thesis, as described in chapter 1 (section 1.3.2.2), is to design and implement a model weaver for AOMDF interaction models. We described in chapter 7 that model weaving is a special case of model transformation, taking multiple models as input and combining them into a single output model. Hence the core of the model weaver can be constructed like a model-to-model transformation. Below, we identify the requirements and challenges associated with the model weaver development, and devise a solution strategy.

8.2.1 Requirements

As for the metamodel, our main requirement for the model weaver is that it should serve its purpose and provide a proper proof-of-concept for weaving of primary interaction models with aspect interaction models. The input models and the woven result must all be well-formed with respect to the new AOMDF metamodel, and it should support both kinds of weaving methods defined in AOMDF (*simpleAspect* and *compositeAspect*). Additionally, we also seek to fulfill the following requirements:

1. **Weaving of underlying structural elements:** Building on the requirement to our metamodel to facilitate *behavior-structure-complete* modeling (see section 8.1.1), we may now require that upon weaving of interaction models, proper weaving of the underlying classes and other structural elements should take place.

2. **Well-formedness validation:** The model weaver should validate that all input and output models are well-formed with respect to the new AOMDF metamodel.

3. **Modularization:** Expecting that the model weaver will turn out to become an overly complex piece of software, use of proper design-time modularization and good exercise of separation of concerns is essential to ensure a comprehensible and evolvable solution.

8.2.1.1 High-level Pseudo Algorithm for Weaving Process

A high-level pseudo-algorithm of the weaving process is beneficial to have understood before embarking on the design and implementation of the model weaver. We state a brief, high-level description of the weaving process to uncover any challenges:

1. **Model loading:** A tagged primary model and referenced aspect models are loaded into the weaver.

2. **Input validation:** The model weaver validates the well-formedness of the tagged primary model with tags containing

weaving instructions. The aspect models referenced by the tags are checked as well.

3. **Deep-copying of primary model elements to target space:** Source models must remain unchanged. A new target model is instantiated and all primary model elements from the source primary model are copied over to the target model.

4. **Extraction of weaving instructions from tagged model:** Weaving instructions in tags are extracted from the source model tagged primary model.

5. **Extraction of aspect advice from aspect models:** Aspect advice from the aspect models are extracted according to aspect model references in the weaving instructions (tags).

6. **Weaving of underlying structural model elements:** Weaving instructions in the tags are parsed for bindings between behavioral elements of the primary model and aspect models. Bindings between structural elements are derived from these and used to weave the aspect models' structural elements into the primary structural elements already in place in the target space. Prior to the weaving, instantiation of the aspect model elements into target space is performed by means of deep-copying.

7. **Weaving of interaction model elements:** Behavioral elements from the aspect models are instantiated into target space by means of deep-copying and woven in between the primary behavioral elements as directed by the weaving instruction extracted from the tags.

8. **Output validation and saving:** Well-formedness of the composed model is evaluated and the model is finally saved. Input models are discarded (i.e. they remain unchanged).

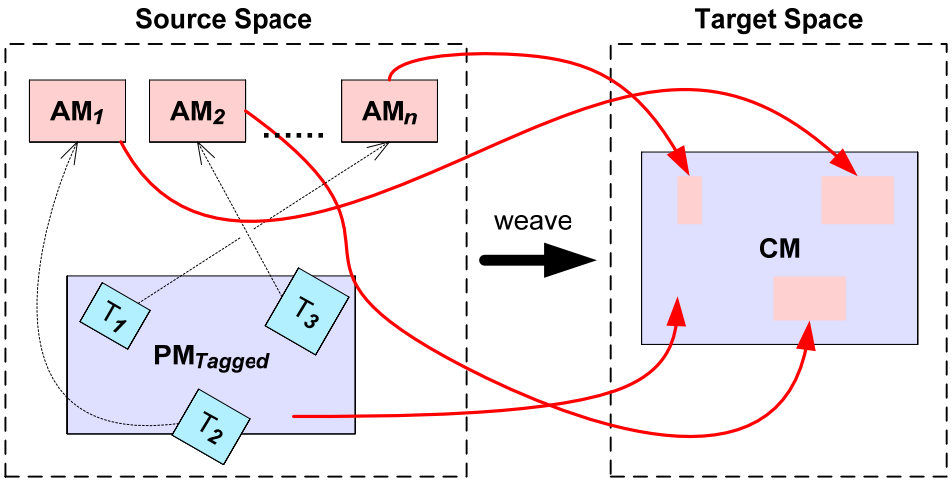

Figure 8-6: Simplified high-level view of the weaving process

A simplified view of the weaving process is illustrated in Figure 8-6. The figure shows a primary model (PM) which has been tagged with a set of weaving instructions (T) that refer to different aspect models (AM) and contain aspect adaptation knowledge supplied by the system designer. The weaving then functions like a transformation producing a composed model (CM) which adapts the features of the aspect models at points where the primary model originally was tagged.

8.2.2 Challenges

8.2.2.1 Heavy-duty model processing

We have earlier discussed the complex nature of the UML2 metamodel, and showed that closely related concrete syntax concepts are distantly related in the abstract syntax. Further, we know by experience that even the simplest interaction models are quite rich in their information content. Given this, and the fact that our model weaver will need to deal with multiple models simultaneously, we expect that significant amounts of processing is required to perform the weaving of interaction models, even in the case of averagely large models. Thus, we classify our model weaver as a heavy-duty [28] transformation and keep this in mind during the weaver design.

8.2.2.2 Nested Weaving

A question that has not been raised in previous work on AOMDF is whether it is feasible to allow the weaving instructions in models to be nested into each other, and thus achieve a hierarchical execution of weavings. This question could be interesting to examine deeper during the model weaver design, however it should be given low priority in order to keep a narrow focus on our goal.

8.2.2.3 Crosscutting Concerns

Apparently, the basic requirements of the model weaver seem to exhibit a crosscutting nature. The weaving process consists of concerns like deep-copying of model elements from source to target space, extraction of weaving instructions tagged into the primary model, extraction of advice from the aspect models, complex model navigation and weaving of model-level data structures. All these concerns definitively crosscut the abstract syntax model and make it challenging to modularize properly and avoid code-tangling in our weaver implementation.

8.2.2.4 Mid-Weaving Traceability Needs

During deep copy creation of model elements from source models to the target space, references between model elements are broken and need to be reconstructed properly in the target space. This can be compared to the act of breaking a jigsaw-puzzle into pieces, and reassembling it in a different location. The only difference is that once the connections between model elements are broken, it is often impossible to reconnect them correctly without

external knowledge. Hence we need to employ some kind of trace mechanism to be able to correctly rebuild references.

Furthermore, since an aspect model may be adopted multiple times during a single weaving execution, we might be able to utilize traces to ensure that we do not unnecessarily duplicate any model elements or accidentally repeat any atomic weaving of two elements.

8.2.3 Solution Strategy

Again, it is crucial that we narrow our scope as much as possible to keep our goal within reach. Considering the identified challenges, the requirements and the pseudo-algorithm described earlier, we prioritize to design and implement only core parts of the model weaver so that we can obtain a solution of high value with respect to our proof-of-concept needs, and uncover any unknown challenges or complexities of interaction model weaving.

As planned, we seek to design our model weaver as a *model-to-model* – or more precisely – a *multi-model-to-model* transformation. The *weave* transformation is defined in **Feil! Fant ikke referansekilden.** below (compare this to Figure 8-6). The inputs to the transformation are a tagged primary model and a set of aspect models. The output is a composed model.

The weave transformation

```
Composed Model =
      weave( Tagged Primary Model, Set<Aspect Model> )
```

Table 8-1: The weave transformation

8.2.3.1 Transformation Classification

Since all the inputs to and the output of our *weave* transformation conform to the same metamodel (as shown in Figure 8-7), we may classify it as an endogenous transformation (in contrast to an exogenous transformation where the source and target models conform to different metamodels).

Furthermore, the transformation does not perform any downshifts in terms of the platform independency of the models. Thus the transformation may also be classified as horizontal (in contrast to vertical transformations that alter the level of abstraction in terms of platform independency). Also, the challenges identified so far by all means indicate that this transformation will be of a complex, heavy-duty nature.

Hence, summarizing these properties with respect to the taxonomy of model transformations presented in [28], the *weave* transformation is a *heavy-duty*, *endogenous*, *horizontal* transformation.

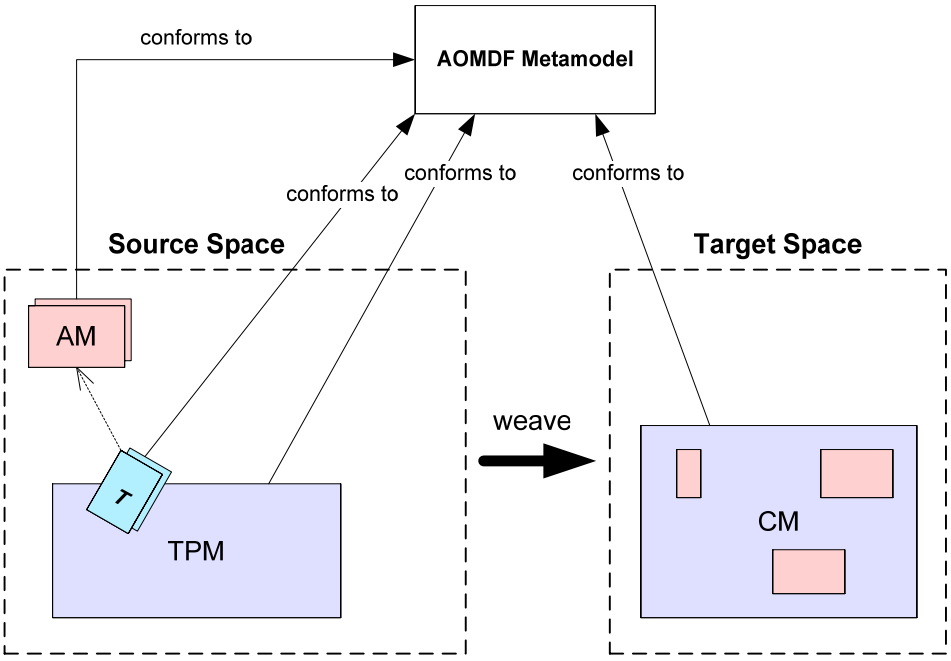

Figure 8-7: Metamodel conformance of weaving inputs and outputs

8.2.3.2 Transformation Language Selection

Most of the currently available tools and languages for building model transformations are found, by experience, to be too simple to support heavy-duty transformations. These languages are usually meant for rapid implementations of rule-based mappings, so whenever the transformation algorithm becomes complex or the mapping is not 1-to-1, proper organization of the transformation code is hard and often not possible. Earlier, we also identified that our transformation deals with several concerns that crosscut the metamodel to which the input and output models conform. Hence, in order to build a properly modularized transformation, we desire a language aware of model concepts and with concern separation features beyond object-orientation.

A language that satisfies our needs is Kermeta, which we introduced in chapter 7. Kermeta is in fact a complete metamodeling and metaprogramming environment in which models as well as programs (including transformations) are perceived as models. We describe the main benefits of selecting Kermeta in the next section.

8.2.3.3 Aspect Oriented Meta-Feature Injection

The main benefits of selecting Kermeta is that we by directly loading the ecore-file containing our metamodel into Kermeta may treat all the concepts (i.e. metaclasses) from our abstract syntax as native objects in the transformation code, and instrument behavior into these in the form of meta-level operations. Kermeta also allows us to completely reopen the metaclasses in our metamodel and inject new structural features into these. Likewise, we may also add new metaclasses within the namespaces of the metamodel packages. In an aspect oriented fashion, the new meta-operations and –

features, defined as Kermeta code, are woven together with the original metamodel by the Kermeta interpreter before execution. (The imported ecore-file containing the metamodel is left unchanged). The result is an executable metamodel. We illustrate this in Figure 8-8.

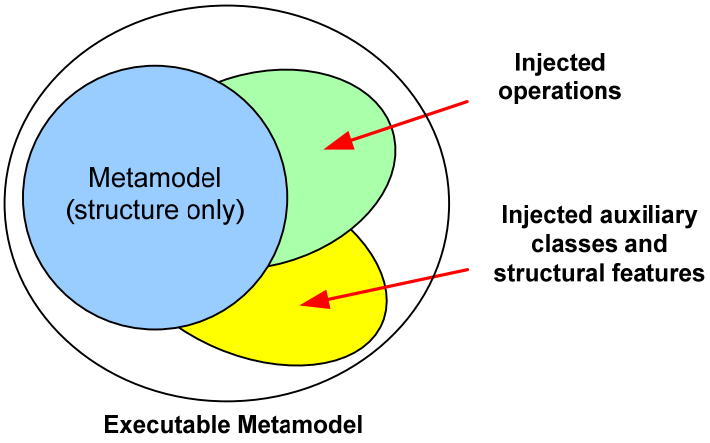

Figure 8-8: Injecting behavior and auxiliary structure into the metamodel

This approach of making the metamodel executable is especially suitable for endogenous transformations as code for many of the lower level concerns like model navigation (i.e. element visiting), deep-copy creation or signature comparison can be encapsulated into the metaclasses and further modularized as aspects. The higher level concerns like weaving and copying of entire model segments can then be programmed at a higher level of abstraction supported by the lower level meta-operations functioning as an API for the higher-level transformation logic. We illustrate this in Figure 8-9.

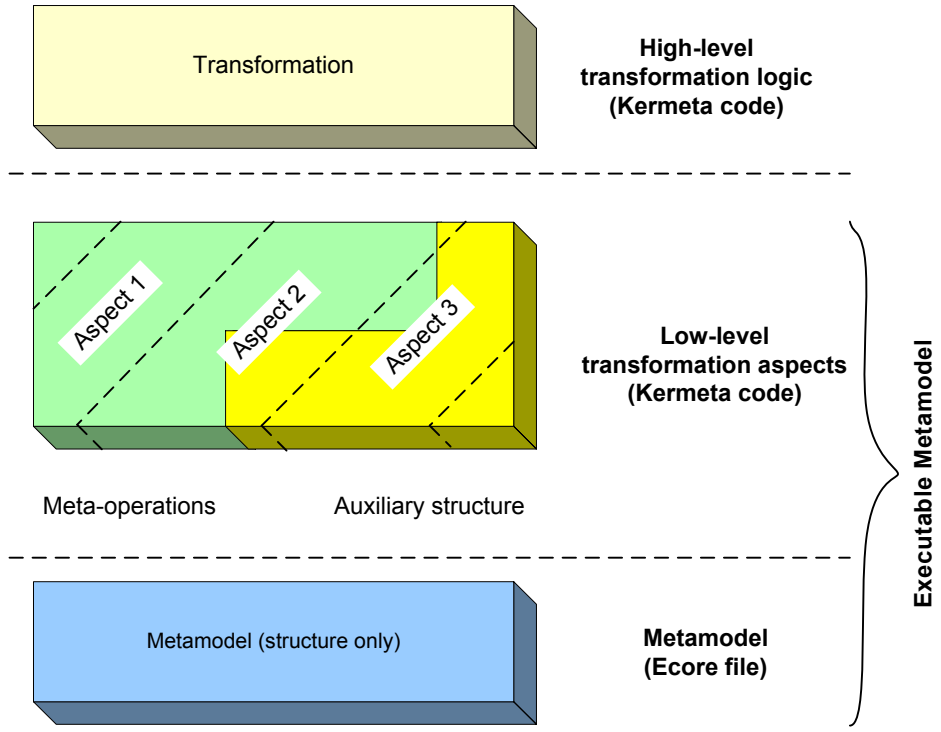

Figure 8-9: Separation of concerns in transformations using Kermeta

8.2.3.4 Signature-Based Weaving of Underlying Class Model

In order to let the weaving of interaction models trigger a weaving of the underlying class models, we can reuse the signature comparison techniques from the proof-of-concept effort on class-model composition in [24]. However, to stay focused on our own goals, we do not fully adopt their approach into our work, but simply utilize the idea of signature-comparisons to evaluate potential collisions during the weaving of structural model elements.

The design and implementation of the interaction model weaver is presented in chapter 10.

8.3 VALIDATION

The scope and time-frame of this thesis only allows for informal validation of our work. As we showed in Figure 1-4 in chapter 1, the validation is performed incrementally during the metamodel and model weaver development since we work in an iterative and incremental fashion.

For validation of the metamodel we instantiate models based on our new abstract syntax concepts and attempt to model the primary and aspect models presented in the example figures in chapter 5.3. If we succeed in properly capturing the example models, we will, informally, have validated the metamodel as fit for purpose. The only challenge related to this is that we will not have any graphical modeling tool available to support us in constructing the models. However, since we are using EMF-based tools, we may utilize the EMF Reflexive Model Editor which is supplied with EMF. This will allow us to utilize a tree-based model editor to define models conforming to our metamodel. These models can eventually be loaded directly into our Kermeta implementation of the *weave*-transformation and the output models can be manually examined to establish conformance or deviations with respect to expected output.

8.4 SUMMARY

In this chapter we have presented requirements for the development of the AOMDF metamodel and the interaction model weaver. Challenges related to each of them have been uncovered and a solution strategy for each has been devised.

Our overall technical solution approach consists of three main steps. Firstly, we develop an abstract syntax for modelling the AOMDF interaction models (aspect interaction models and weaving instructions in primary models). The abstract syntax is developed in the form of a metamodel that orthogonally extends UML2. Secondly, we design and implement a model weaver as a horizontal, endogenous model transformation – conforming to our metamodel. Unlike traditional ways of implementing model transformations, the model

weaver is realized by injecting behavior and auxiliary features into our metamodel, in an aspect oriented fashion, using Kermeta. This way the concerns of the transformation are better modularized and the higher-level transformation logic becomes easier to implement.

Finally we outline simple test cases to informally validate the model weaver.

The subsequent chapters respectively present the metamodel and the model weaver.

9 Metamodel for AOMDF Interaction Weaving

The first step in our work was to model the abstract syntax of AOMDF interaction models in the form of a metamodel as described in chapter 1.3.2. A solution strategy for this was devised in chapter 8.1. In this chapter we present the metamodel we have developed based on our strategy.

9.1 PACKAGE STRUCTURE AND REUSE

Reflecting the layers shown in Figure 8-5, the metamodel is structured into three packages as illustrated in Figure 9-1.

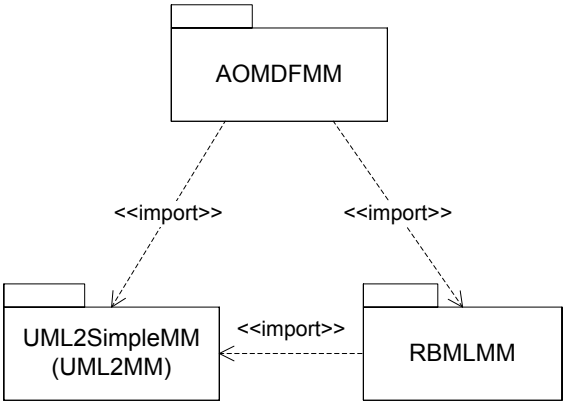

Figure 9-1: Metamodel package organization

According to our strategy we have modeled a simplified subset of the UML2 metamodel in the UML2SimpleMM-package. This is imported and extended by the RBMLMM[4]-package to provide role- and pattern specification concepts. The AOMDFMM-package imports both UML2SimpleMM and RBMLMM, and introduces the new modeling concepts.

In the subsequent sections, we present the subpackages and concepts modeled into each of these three packages.

[4] The RBMLMM-package is also simplified in the sense that it can only contain role-concepts for the UML2-subset found in the imported UML2SimpleMM. For the sake of simplicity we leave its name unchanged.

9.2 UML2SIMPLE

The *UML2SimpleMM* package contains a simplified metamodel for UML2, created by examining the UML2 abstract syntax and carefully picking out individual elements and segments of elements required for expressing basic UML2 class- and interaction models (i.e. simple class- and sequence diagrams). The selected elements are connected together mostly in the same way as they are in the original UML2, however we have made simplifications in the form of eliminating inheritance layers that do not have any purpose in our simplified UML2 metamodel. This process is illustrated in Figure 9-2.

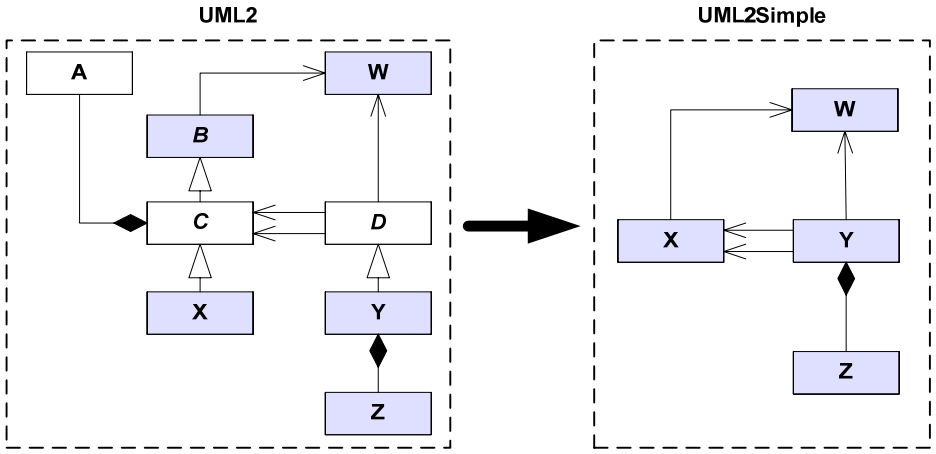

Figure 9-2: Simplification by subset selection and inheritance flattening

As the above figure suggests, the conceptual integrity of UML2 abstract syntax is preserved in UML2Simple, and hence any extensions orthogonal to UML2Simple will also be orthogonal to the UML2 abstract syntax. (We do however introduce a minor modification to a UML2 concept in section 9.2.3 in order fit one of our later extensions properly).

UML2SimpleMM is made up by the three subpackages *Classes*, *Structures* and *Interactions* as shown in Figure 9-3. Together, these three packages allow us to express basic UML2 class- and interaction models using an abstract syntax that is easier to navigate and experiment with.

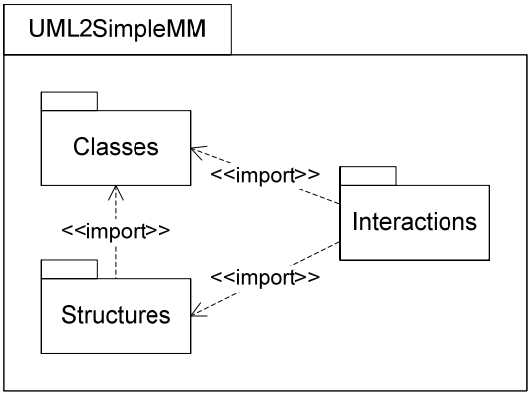

Figure 9-3: Subpackages in UML2SimpleMM

Readers interested in detailed descriptions of the classes contained in *UML2SimpleMM* should refer to the UML2 specification.

9.2.1 Classes

The *Classes* subpackage captures the concepts needed for basic structural modeling (i.e. UML2 class-models) and the root-elements of the UML2 abstract syntax. Figure 9-4 shows the well-known concepts *Model*, *Package* and *Classifier*. *Element* is the root-element for all meta-classes in the UML2 metamodel.

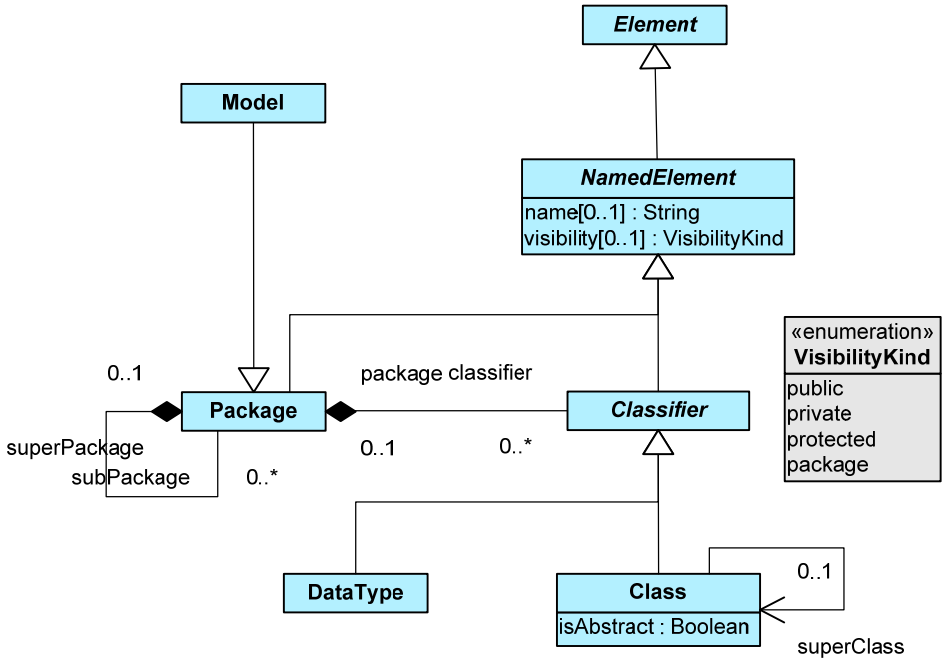

Figure 9-4: Models, Packages and Classifiers in UML2SimpleMM

Figure 9-5 (on next page) describes the concepts *Class*, *Attribute*, *Reference* and *Operation* that as we know them from ordinary UML2 class diagrams. The only element that may seem a bit unfamiliar for the readers not familiar with the UML2 specification is the *ValueSpecification*, which here is used to model a placeholder for values, such as default values for operation parameters or (as we will observe later) for arguments passed during an operation call.

9.2.2 Structures

In the *Structures* subpackage, shown in Figure 9-6 (on next page), we have included the concept *Property* and its generalization *ConnectableElement*, which is used to model an instance (or a set of instances) of a classifier (for example a *part* in a *composite structure*). These concepts are necessary for the connection between class- and interaction models as *Lifeline*s in interaction models represent *ConnectableElement*s. In other words this concept is used to model the participation of a class instance in an interaction.

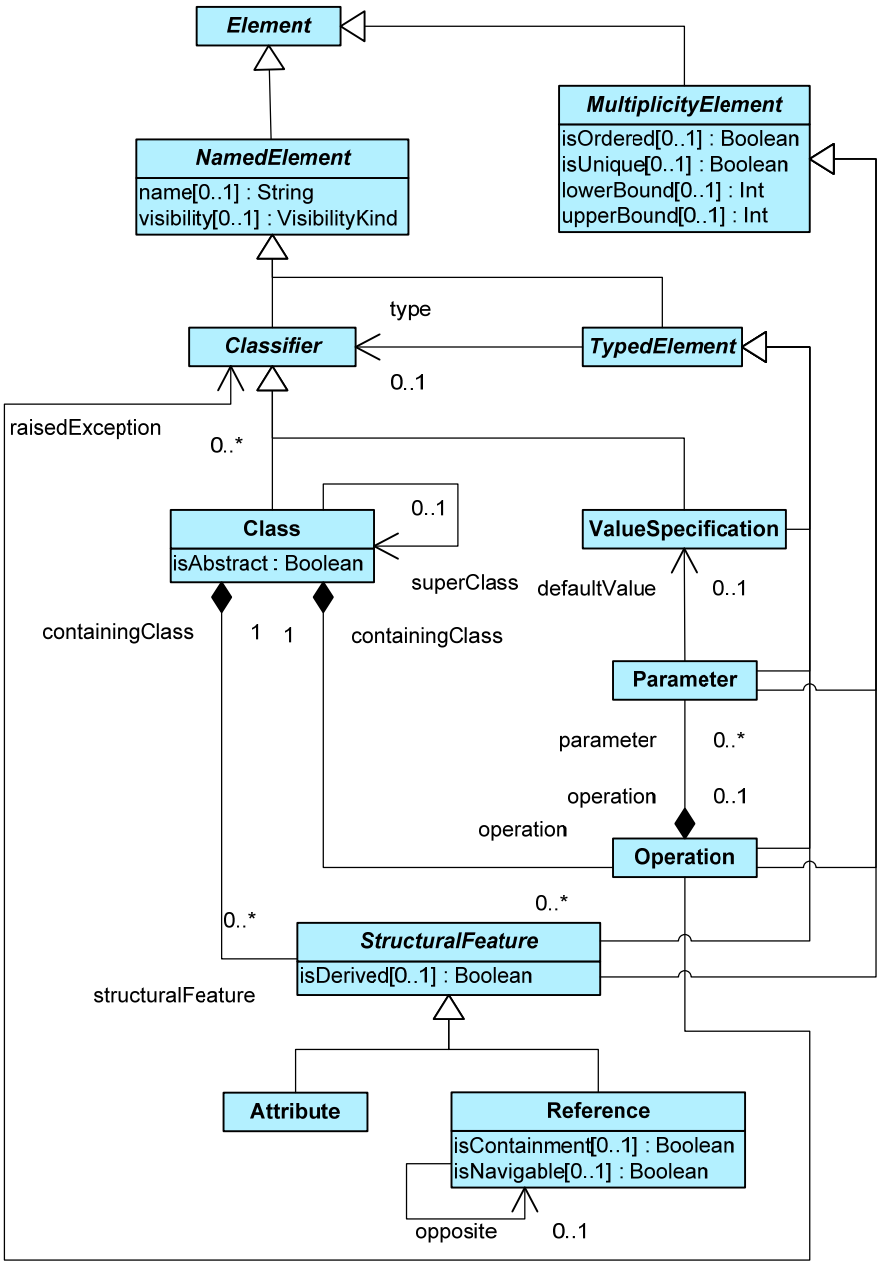

Figure 9-5: Classes, StructuralFeatures and Operations in UML2SimpleMM

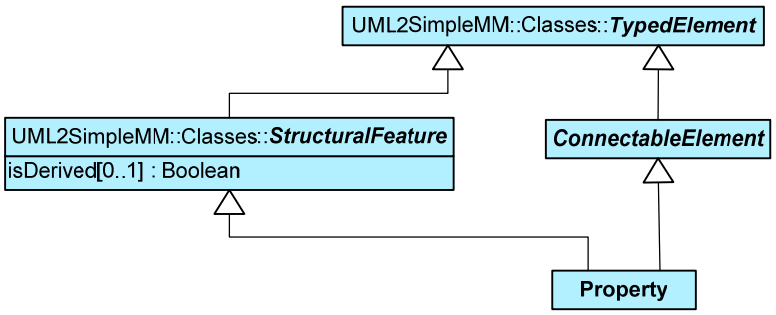

Figure 9-6: ConnectableElements and Properties in UML2SimpleMM

9.2.3 Interactions

The *Interactions* subpackage captures the concepts required for basic behavioral modeling (i.e. UML2 interaction / sequence models).

Figure 9-7 shows the first-class element *Interaction*, and the main components in an interaction model: *Message*s, *Lifeline*s and *InteractionFragment*s. Not shown is this figure is the fact that *Interaction*, in accordance with the UML specification, is a specialization of *Class* (from *UML2SimpleMM::Classes*). This allows *Interaction*s to be contained in a *Package*.

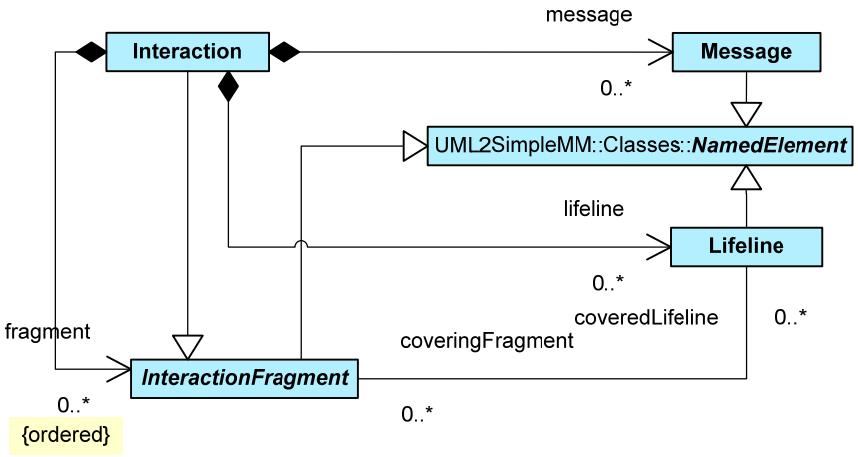

Figure 9-7: Interactions in UML2SimpleMM

Figure 9-8 details the *Lifeline*-view, while Figure 9-9 (on next page) details the *Message*- and *MessageEnd*-view of the *Interactions* subpackage.

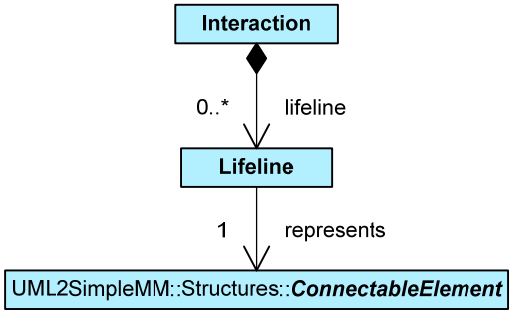

Figure 9-8: Lifelines in UML2SimpleMM

Figure 9-10 (on page 65) shows *InteractionFragments* and related concepts. This part is very central for the rest of our work so we describe it thoroughly.

9.2.3.1 Interaction Fragments

An *InteractionFragment* represents a piece of an *Interaction*, and is considered to be an Interaction of its own – either atomic or composite. Hence we see in Figure 9-10 (on page 65) that an *Interaction* is actually a specialization of *InteractionFragment*. However, the important thing to note is that the set of fragments contained by an *Interation* is ordered, i.e. the contained *InteractionFragment*s are stored in a sequence. This is how time-

sequence information – which is visually available in sequence diagrams – is retained in interaction models.

UML2 defines several atomic *InteractionFragment*s used in various kinds of interaction diagrams. We have only included the two we need for modeling basic sequence diagrams like the base models in the AOMDF-examples presented in chapter 5.3. These are *MessageOccurenceSpecification* and *BehaviourExecutionSpecification*.

Message events like invocation or reception of operation calls are specified as *MessageOccurenceSpecification*, which is subject to dual inheritance from both *InteractionFragment* and *MessageEnd*. This way a *Message* can be connected to two *MessageOccurenceSpecification*s (see Figure 9-9). The execution of behavior-units on *Lifeline*s is specified as *BehaviourExecutionSpecification*. Message events marking the start and finish of the behavior-units are captured by the references *start* and *finish*.

A composite (i.e. non-atomic) *InteractionFragment* is specified as a *CombinedFragment* containing an ordered set of *InteractionOperand*s, where each operand in turn contains an ordered set of *InteractionFragment*s. This mechanism allows for infinite levels of *InteractionFragment*-nesting.

While a *CombinedFragment* is defined by an *InteractionOperator* in the UML2 specification, we have performed a minor modification at this point in UML2SimpleMM and introduced *StandardCombinedFragment* as a specialization of *CombinedFragment*. *StandardCombinedFragment* now represents the ordinary *CombinedFragment*. The motivation behind this decision is that this point is suitable for introducing the new *InteractionFragment*-concepts later on in the *AOMDFMM*-package. The modification is highlighted with a note in Figure 9-10.

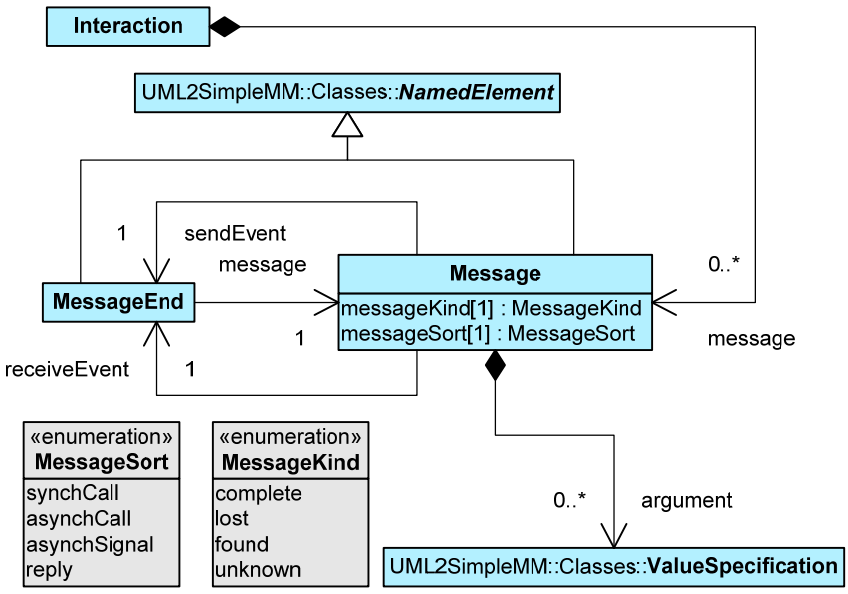

Figure 9-9: Messages and MessageEnds in UML2SimpleMM

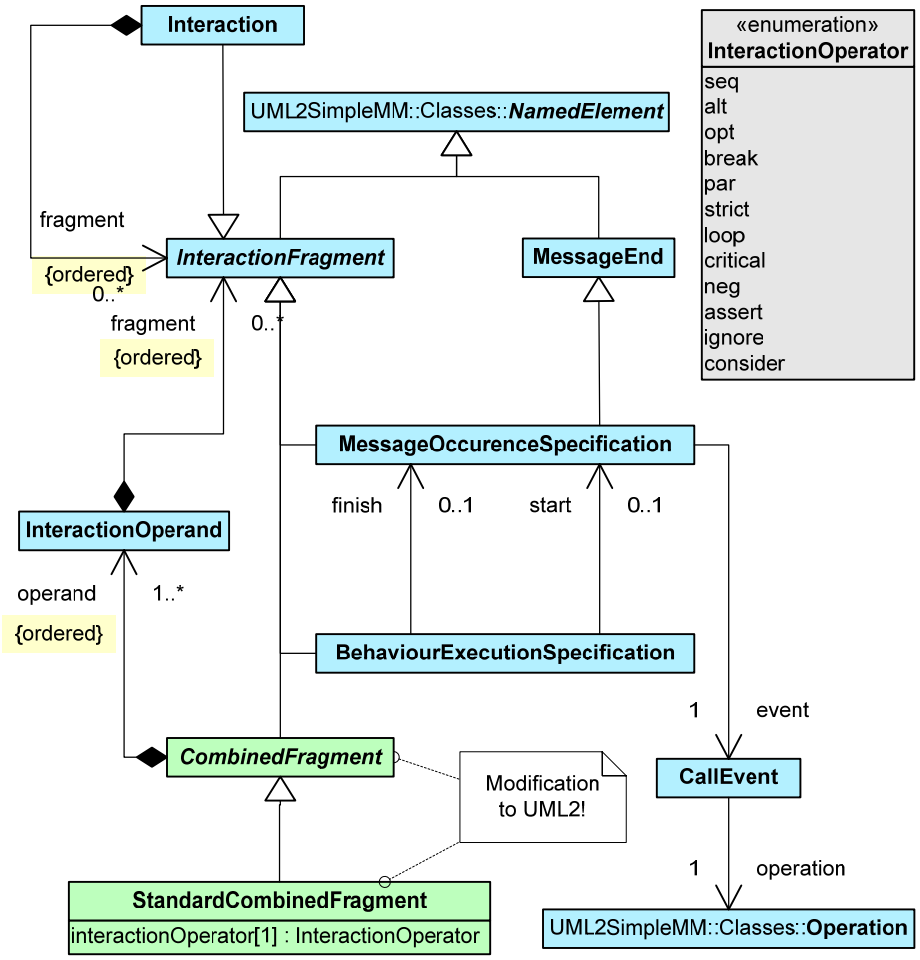

Figure 9-10: InteractionFragments and -Operands in UML2SimpleMM

Consideration of more advanced concepts from the UML2 Interactions' abstract syntax like *Gate*, *InteractionUse* and *PartDecomposition* [19] are left outside of the scope of our work.

9.3 RBML

RBML is a complete modeling language (with defined notation and semantics) and is suitable for specification of model-patterns. The RBML abstract syntax is defined as a mere specialization of the concepts in the abstract syntax of UML, as illustrated in Figure 9-11.

The *RBMLMM* package contains a tailored subset of the RBML metamodel. We have included the RBML-concepts required for representing the role-elements observed in the aspect models in the AOMDF-examples presented in chapter 5.3. Like *UML2SimpleMM*, the *RBMLMM* package is split into the three subpackages *Classes*, *Structures* and *Interactions*. This is shown in Figure 9-12.

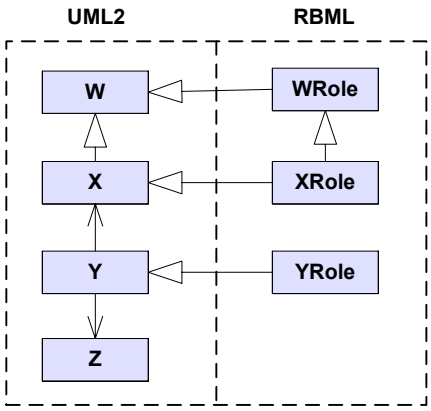

Figure 9-11: RBML specializes UML2

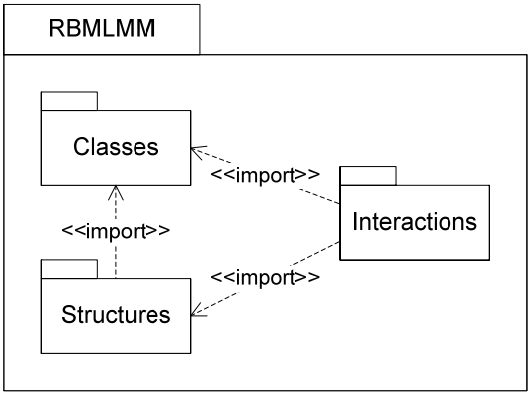

Figure 9-12: Subpackages in RBMLMM

9.3.1 Classes

Figure 9-13 shows the contents *RBMLMM::Classes*. The two essential concepts here are the *ClassRole* and the *ValueSpecification*. The first is used to specify a class-role in aspect class models, while the latter is used to specify argument-roles in aspect interaction model.

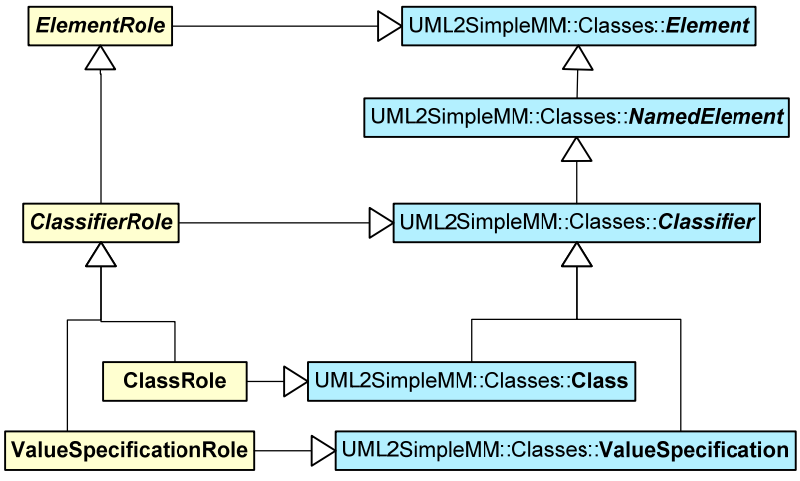

Figure 9-13: ClassifierRoles in RBMLMM

9.3.2 Structures

Figure 9-14 shows the contents of *RBMLMM::Structures*, containing the *PropertyRole* and *ConnectableElementRole* concepts used for specifying classifier-instance-roles.

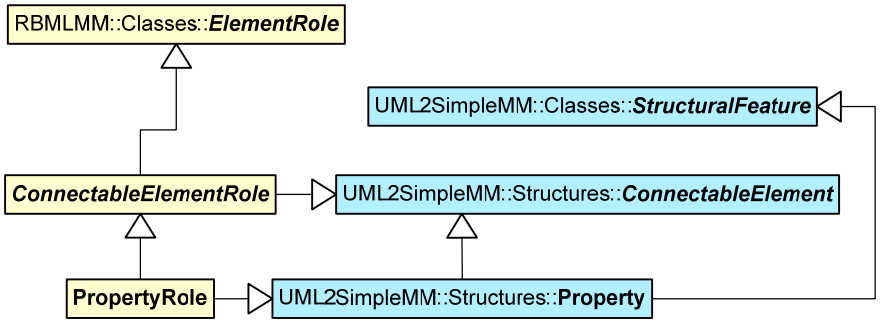

Figure 9-14: ConnectableElementRoles in RBMLMM

9.3.3 Interactions

The *RBMLMM::Interactions* subpackage, shown in Figure 9-15, contains *LifelineRole* and *MessageRole*, used for specifying lifeline-roles and message-roles respectively in aspect interactions.

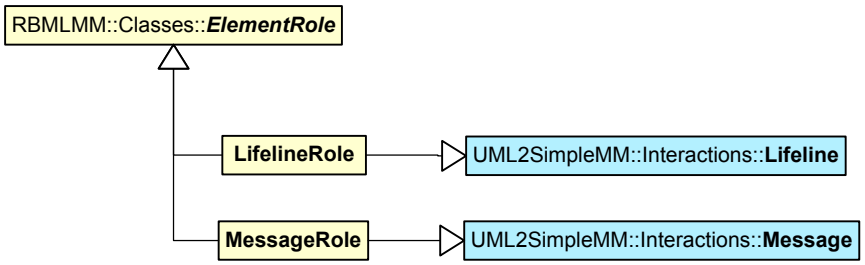

Figure 9-15: Lifeline- and MessageRoles in RBMLMM

9.4 AOMDF

The *AOMDFMM* package, representing the uppermost layer in Figure 8-5, is where we introduce the new modeling concepts required for tagging and weaving. As shown in Figure 9-16, we structure it into four subpackages:

- *AOMDFMM::Models*
- *AOMDFMM::Classes*
- *AOMDFMM::Interactions*
- *AOMDFMM::Bindings*

The subpackages are separately described in the next sections.

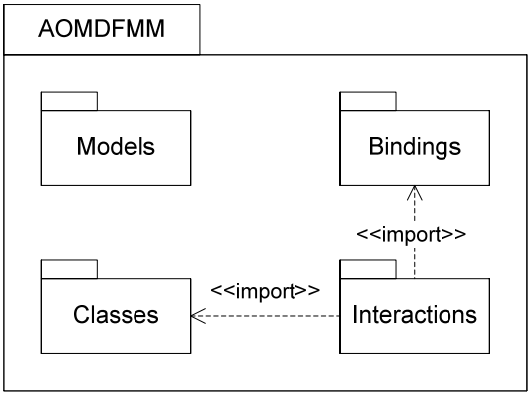

Figure 9-16: Subpackages in AOMDFMM

9.4.1 Models

In the *Models* subpackage, we have specialized the concept *Model* from *UML2SimpleMM* into two different subtypes that we need to distinguish between in AOMDF – *PrimaryModel* and *AspectModel* as shown in Figure 9-17.

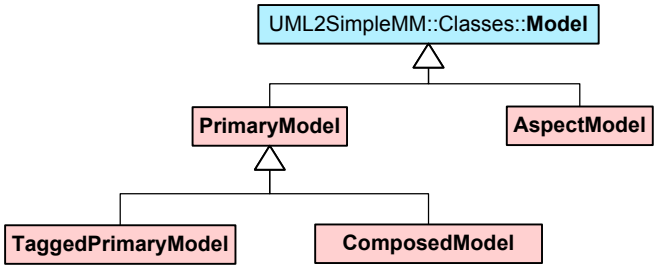

Figure 9-17: New subtypes of Model

PrimaryModel is further specialized into *TaggedPrimaryModel* and *ComposedPrimaryModel*. A *TaggedPrimaryModel* is a *PrimaryModel* which also contains tags with instructions for the weaving process. A *ComposedModel* is a *PrimaryModel* resulting from a weaving (i.e. free of tags).

These concepts are useful in order to keep a precise terminology in the framework. While the existence of these concepts is not necessary for the proof-of-concept development of our model weaver, they will even though be of significant importance in the complete AOMDF modeling toolkit described in section 1.3.1 in our Introduction.

Having these concepts in our metamodel also allows us to state that the model weaver will take a *TaggedPrimaryModel* and set of *AspectModel*s as input and produce a *ComposedModel* as output. The responsibility for transforming a *PrimaryModel* into a *TaggedPrimaryModel* should be assigned to the Modeling Tool component, which is not developed within the scope of this thesis (see Figure 1-3 on page 4).

9.4.2 Classes

The *Classes* subpackage is left empty for now as we plan to let the weaving of interaction-models drive the weaving of the underlying class-model. Thus we need no new elements here. However, this package will most probably be populated with new elements in the future, when we attempt to integrate the previous proof-of-concept work on class-model weaving [3, 24] with our work.

9.4.3 Interactions

The *Interactions* subpackage in *AOMDFMM* contains extensions to *UML2SimpleMM::Interactions*. The extensions are grouped into two segments of new concepts, one for capturing the advice in aspect models, and one capturing the *weaving points*[5] containing the weaving instructions in primary models. We describe each segment below.

9.4.3.1 Aspect Interactions and Advice

As shown in Figure 9-18, we define two new concepts, and related enumerations, to express aspect interaction models and the advice within these.

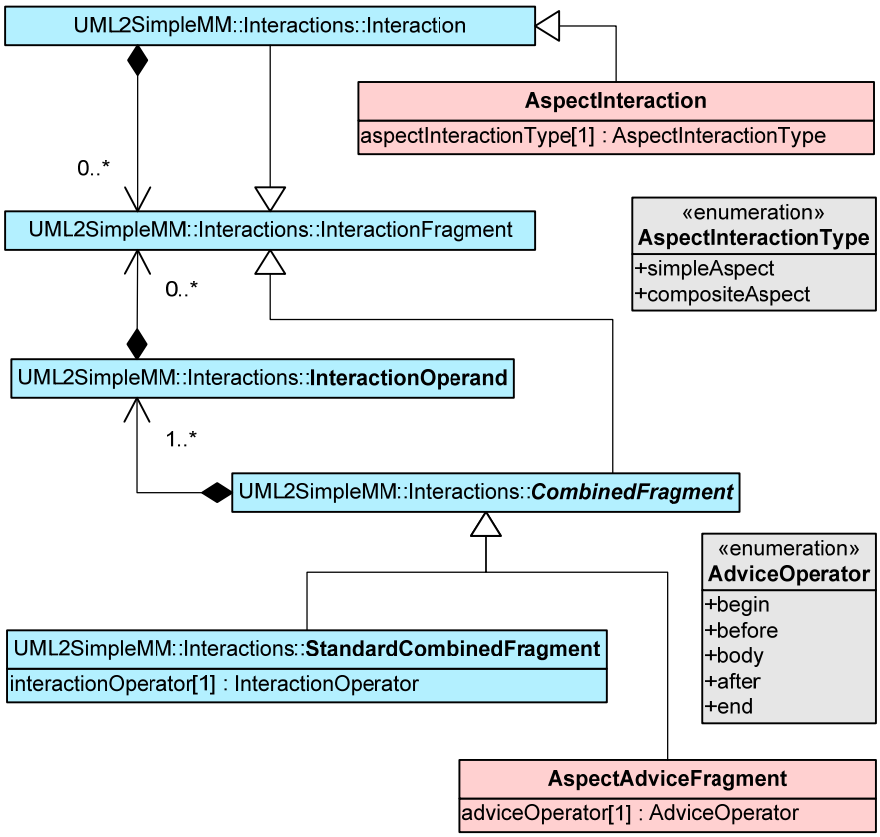

Figure 9-18: AspectInteraction and AspectAdviceFragment

[5] The weaving instructions tagged into primary models have simply been referred to as *tags* by previous work on AOMDF. We, however, choose to call them *weaving points* in our work.

Firstly, we introduce *AspectInteraction* as a specialization of the ordinary *Interaction* from *UML2SimpleMM*. *AspectInteraction*s have an *aspectInteractionType* which may take the value *simpleAspect* or *compositeAspect*. An *AspectInteraction* of type *simpleAspect* is used to model an interaction sequence that upon weaving will be entirely adopted and inserted into an interaction sequence in a primary model. An *AspectInteraction* of type *compositeAspect* is used to model an interaction sequence that upon weaving will be instrumented in a more intricate manner within a specified subsequence of an interaction sequence.

In other words, a *simpleAspect* is like a simple piece of behavior that is directly injected into certain places in a primary interaction, while a *compositeAspect* is more like an aspect as known from aspect oriented programming languages like AspectJ [56].

For modeling of advice within an *AspectInteraction* of type *compositeAspect*, we introduce *AspectAdviceFragment* as specialization of *CombinedFragment*. Recall here that we – as described in section 9.2.3.1 and Figure 9-10 – performed a minor modification in *UML2SimpleMM* at this point in order to make room for new composite *InteractionFragment*s like *AspectAdviceFragment*.

Defined this way *AspectAdviceFragments* work like *CombinedFragments* and may contain any of the atomic as well as composite *InteractionFragments* available in *UML2SimpleMM*. (Well-formedness contraints in section 9.5 rule out the possibility of nesting *AspectAdviceFragments*).

To govern the instrumentation of their contained *InteractionFragment*s into a primary interaction, *AspectAdviceFragment*s are defined with an *AdviceOperator*. The valid values for this operator are described in Table 9-1.

AdviceOperator	***Semantics***
begin	Insert at the beginning of the subsequence being instrumented
before	Insert before every element in the subsequence being instrumented
body	Embed into elements of subsequence being instrumented
after	Insert after every element in the subsequence being instrumented
end	Insert at the end of the subsequence being instrumented

Table 9-1: AdviceOperators and their semantics

*AspectInteraction*s of both types may contain ordinary *Lifeline*s, *Message*s and *InteractionFragments*. However, all messages in a *compositeAspect* must be contained by an *AspectAdviceFragment*. Furthermore, only one *AspectAdviceFragment* per *AdviceOperator* is allowed in an *AspectInteraction*

of type *compositeAspect*. No *AspectAdviceFragment*s are allowed in a *simpleAspect*.

Certainly, *AspectInteraction*s must also contain at least one *LifelineRole*, that will be populated by a *Lifeline* from the primary model upon weaving. In the *simpleAspect* case there must be exactly one *LifelineRole* present in the *AspectInteraction*. In the *compositeAspect* case one or more *LifelineRole*s may exist. According to the semantics of *AdviceOperator*, only *AspectAdviceFragment*s defined with the operator *body* may contain *MessageRoles*.

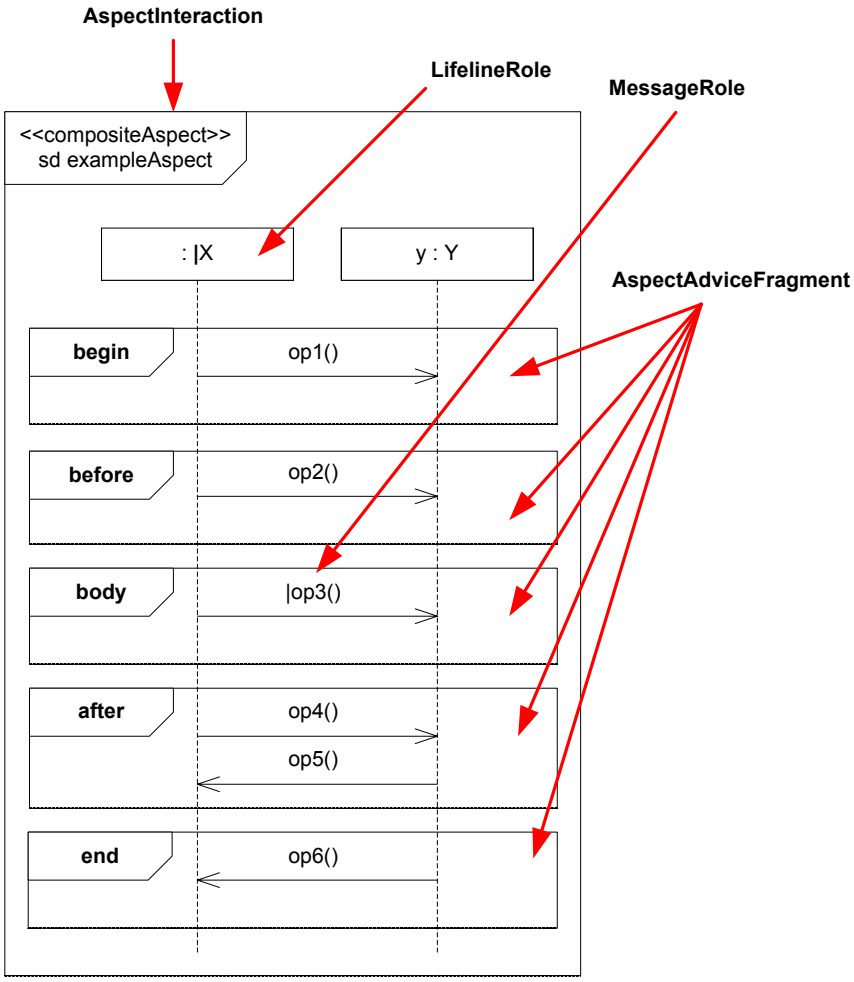

Figure 9-19: Concrete syntax for aspect advice

Figure 9-19 illustrates the suggested concrete syntax for *AspectInteraction*s and *AspectAdviceFragment*s, and hopefully clarifies the above presentation of the abstract syntax. (Only minor modifications to the concrete syntax have been made here compared to the previous proposals for the concrete syntax).

9.4.3.2 Weaving Points

Weaving of primary- and aspect models requires a primary model, a set of aspect models and some information about where in the primary model the various aspects should be woven in and how they should be adapted. This

information is tagged into the primary model (which then becomes a *TaggedPrimaryModel*). Previous work on AOMDF has referred to this information as *tags*. However, the word *tag* is bloated in both the English language in general and in various terminologies of model-based software development. Thus, we rather choose to use the term *weaving point,* which we feel is more precise than *tag*, to talk about the information containing weaving instructions. (The sentence "we tag a primary model with weaving points" sounds more precise than "we tag a primary model with tags"). By a weaving point we mean a certain point (or subsequence) in an interaction where some aspect behavior will be injected (or woven). This is analogous to *join points* in aspect oriented programming languages.

In Figure 9-20, we introduce *AspectWeavingPointFragment* as the modeling concept for specifying a weaving point. Since a weaving point marks a point or a subsequence in an interaction sequence we define *AspectWeavingPointFragment* as a specialization of *CombinedFragment* (in the same way as we did with *AspectAdviceFragment* earlier). This way it functions as a composite *InteractionFragment* embracing a subsequence of *InteractionFragments* in a primary model.

AspectWeavingPointFragment is defined as an abstract supertype, and specialized into two subtypes *SimpleWeavingFragment* and *CompositeWeavingFragment*. The first is used to specify weaving points for adapting aspect interactions of type *simpleAspect*, the latter to specify weaving points for adapting aspect interactions of type *compositeAspect*. Hence, the *SimpleWeavingFragment* – only marking a certain point – embraces an empty subsequence of the primary interaction sequence. We clarify this in Figure 9-21, where we propose the concrete syntax for weaving points (as an update to the concrete syntax proposed for *tags* in the previous work on AOMDF).

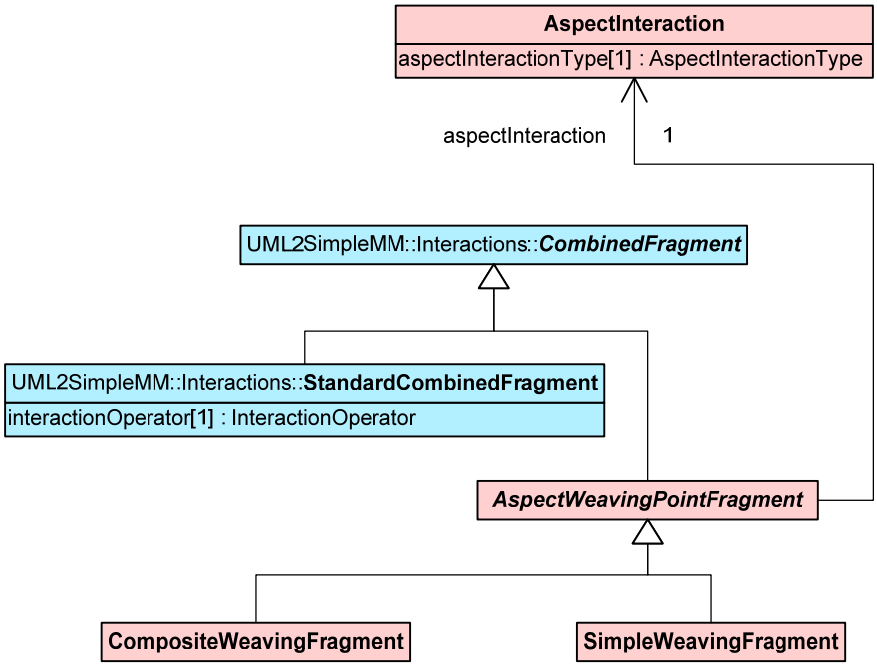

Figure 9-20: AspectWeavingPointFragments

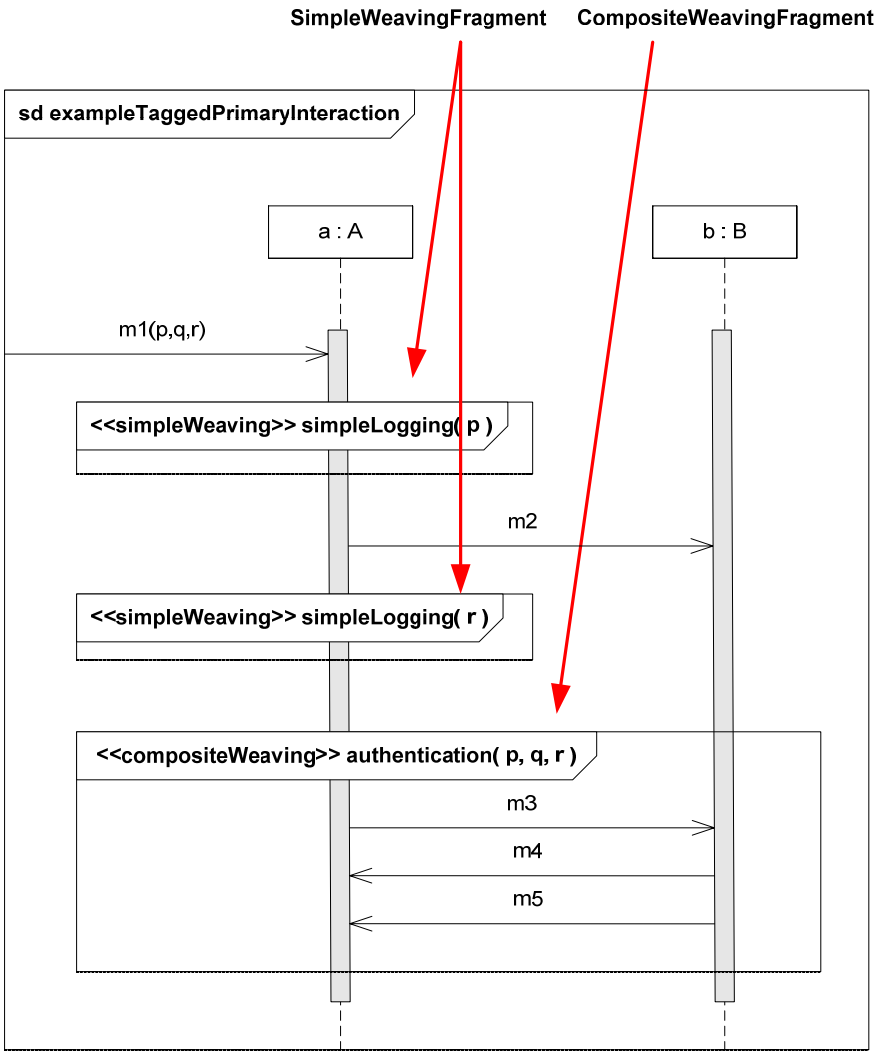

Figure 9-21: Concrete syntax for Weaving Points

Weaving points provide information on where in the primary interaction to weave some aspect behavior. However, the weaving process also requires the knowledge of which aspect interaction to weave at a specific weaving point and how to populate any role elements in the aspect interaction – i.e. how the given aspect is to be adapted at that specific weaving point. Modeling concepts for capturing this knowledge are easily designed as properties of a weaving point as shown in Figure 9-22 (on next page).

Knowledge of what to weave into the weaving point is captured by letting *AspectWeavingPointFragment* reference an *AspectInteraction*. The idea behind this is easily understood if one imagines the scenario where a developer using our modeling toolkit (described in chapter 1) tags his primary model with weaving points and selects which aspect interaction to weave at these points. The modeling tool could then add the reference from the *AspectWeavingPointFragment* to the selected *AspectInteraction* (residing in an *AspectModel* kept in the aspect repository component – recall Figure 1-2 on page 3).

The knowledge of how the various roles in the *AspectInteraction* should be populated during adaptation is captured by letting weaving points contain a set of binding specifications that bind together elements in the primary model with role-elements in the aspect model. Since the population of a lifeline role is required during adaptation of both types of aspect interactions, we let *AspectWeavingPointFragment* contain a set of *LifelineBindingSpecification*s, which in turn contains a set of *ArgumentBindingSpecification*s. *CompositeWeavingFragment* may additionally contain a set of *MessageBindingSpecification*s and a set of *ExclusionSpecification*s since these are only relevant in the *compositeAspect* situation.

As previously mentioned, the binding specifications have been devoted a subpackage of their own and are described in detail in the next section. Previous work on AOMDF has suggested a textual concrete syntax for expressing the binding specifications. However, we believe that some sort of property editor inside the modeling tool would be better to utilize instead of the textual form. Hence we leave any reconsideration of this for future work.

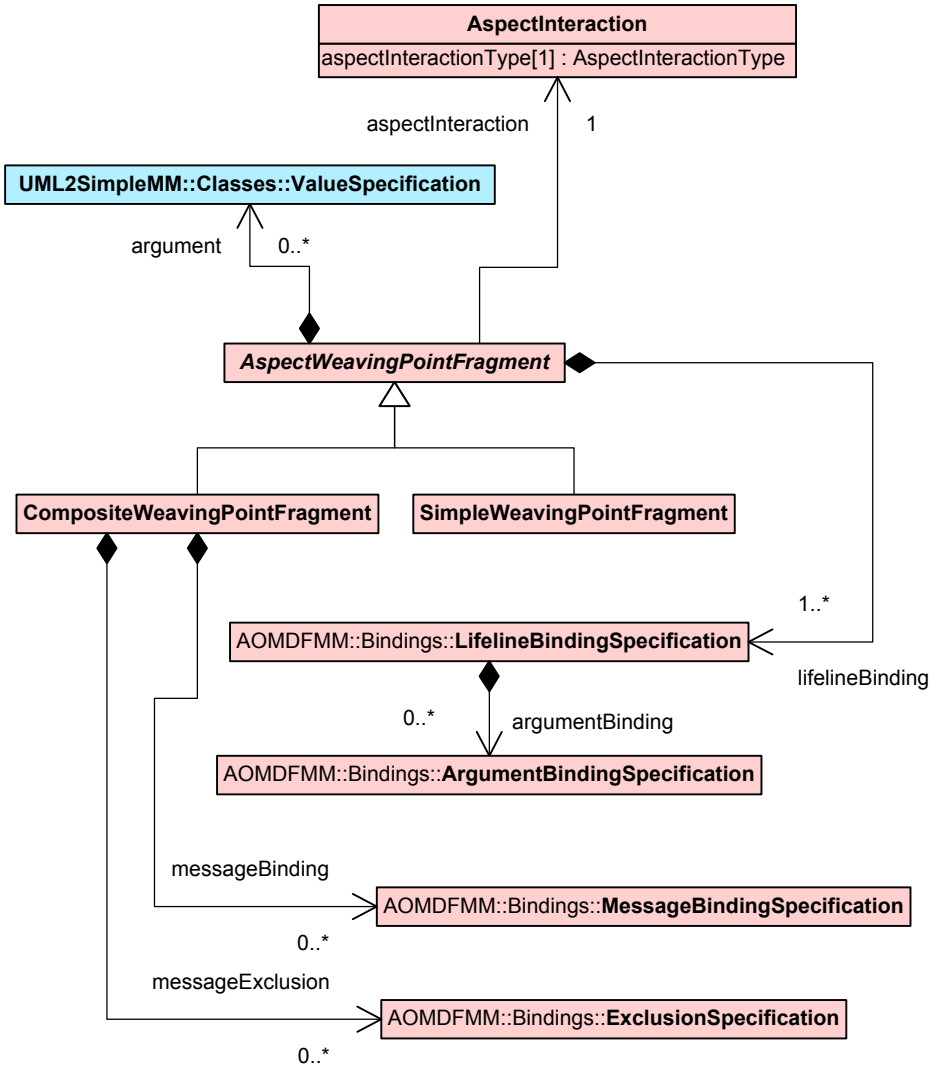

Figure 9-22: Simple- and CompositeWeavingFragment

9.4.4 Bindings

The *Bindings* subpackage contains the metamodel elements for capturing the part of weaving instructions that deal with the population of role elements in aspect models by elements of a primary model – i.e. the knowledge required for adaptation of an aspect into a weaving point in the base, primary model. As introduced in the previous subsection, these concepts are called binding specifications.

9.4.4.1 General Element-Binding

In order to satisfy our requirement of extendibility, we first define an abstract supertype *ElementBindingSpecification* by extending *Element* from *UML2SimpleMM* (*Element* is the root element in the UML2 metamodel). *ElementBindingSpecification* binds an *Element* and an *ElementRole* from *RBMLMM* through the references *primary* and *role* as shown in Figure 9-23.

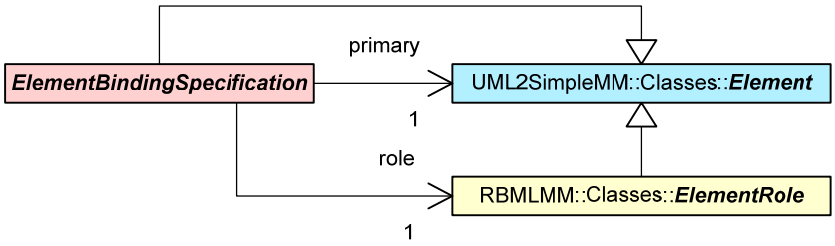

Figure 9-23: ElementBindingSpecification

All other types of binding specifications can now be defined as specializations of this construct (i.e. by specializing *ElementBindingSpecification* and its two references). A complete overview of the binding specification hierarchy is shown in Figure 9-26 on page 77. Future extensions of AOMDF that may address weaving of other kinds of UML2 models (such as composite structure diagrams or state charts) can easily extend this hierarchy to facilitate their needs for specifying bindings.

9.4.4.2 Lifeline- and Argument Binding

In Figure 9-24 we show the specialization of our *ElementBindingSpecification* construct to *LifelineBindingSpecification* and *ArgumentBindingSpecification*. The first is used to bind a Lifeline with a LifelineRole, the latter to bind a *ValueSpecification* with a *ValueSpecificationRole*. By letting *LifelineBindingSpecification* contain a set of *ArgumentBindingSpecification*s we are able to model the binding of arguments on an incoming message on a lifeline (in a primary model) to arguments of an outgoing message on a lifeline role (in an aspect model).

9.4.4.3 Message Binding and Weaving Exclusion

Figure 9-25 shows *MessageBindingSpecification* and *ExclusionSpecification*. *MessageBindingSpecification* is, as shown earlier in Figure 9-22, used together with *CompositeWeavingFragment* to model the binding of the

*Message*s embraced by a *CompositeWeavingFragment* to *MessageRole*s in the referenced *AspectInteraction*. *ExclusionSpecification* is used to exclude a *Message* embraced by the *CompositeWeavingFragment* from the weaving process, so that it is left totally untouched during the aspect adaptation.

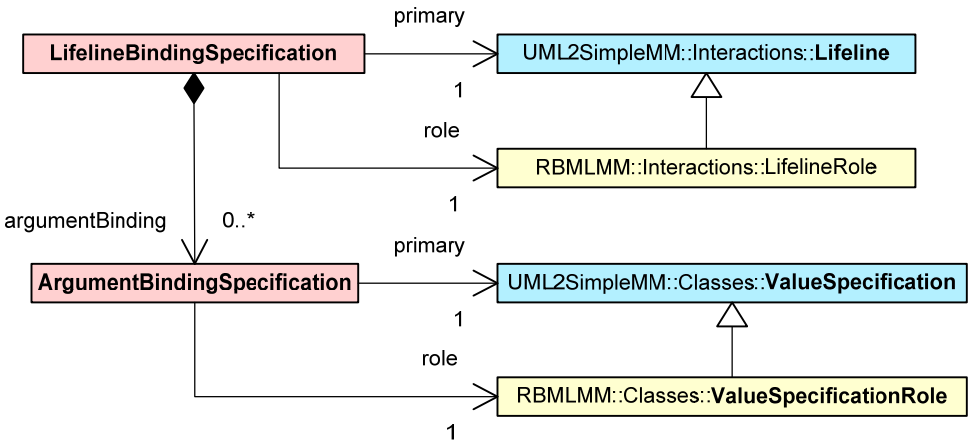

Figure 9-24: Lifeline- and ArgumentBindingSpecification

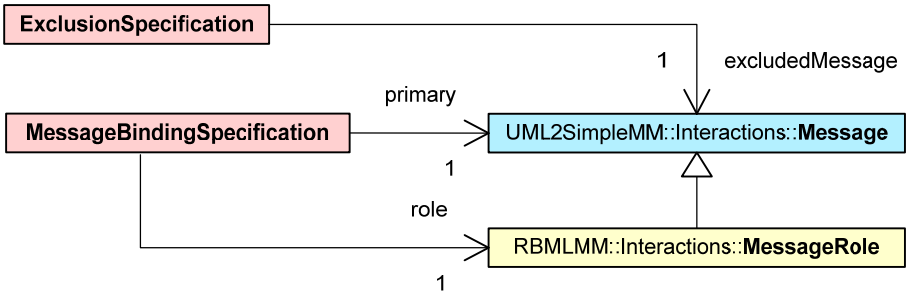

Figure 9-25: MessageBindingSpecification and ExclusionSpecification

9.4.4.4 Derivable Classifier- and ConnectableElement Bindings

We have previously stated that we wish to employ a solution in which the supplied instructions for the interaction model weaving also drive a weaving of the underlying structural elements. Recall from *UML2SimpleMM*, where we showed that a *Lifeline* represents a *ConnectableElement* (or its subtype *Property*) that is typed by a *Classifier* (or its subtype *Class*). We envision that once a binding between a *Lifeline* and *LifelineRol*e is instantiated, we could in fact derive from it a binding between a *ConnectableElement* and a *ConnectableElementRole*. From this derived binding we could further derive the binding between a *Classifier* and a *ClassifierRole*, and thus end up with a chain that lets us derive weaving instructions for class models from the weaving point data.

Hence, we include in our metamodel the concepts for specifying binding of selected structural elements with their role-variants from (*RBMLMM*), and define four new specializations of our general binding specification construct; *ClassifierBindingSpecification* and *ConnectableElementBindingSpecification*

as abstract supertypes, and *ClassBindingSpecification* and *PropertyBindingSpecification* as their respective subtypes. *LifelineBinding-*, *ConnectableElementBinding-* and *ClassifierBindingSpecification* are associated through the references *representedElementBinding* and *typeBinding*, and are shown in Figure 9-26 together with an overview of the binding specification hierarchy.

Once a *LifelineBindingSpecification* is instantiated, a *ConnectableElementBindingSpecification* and *ClassifierBindingSpecification* can, for example, be generated by the modeling tool or as preparation step in that the model weaver performs before initiating the weaving.

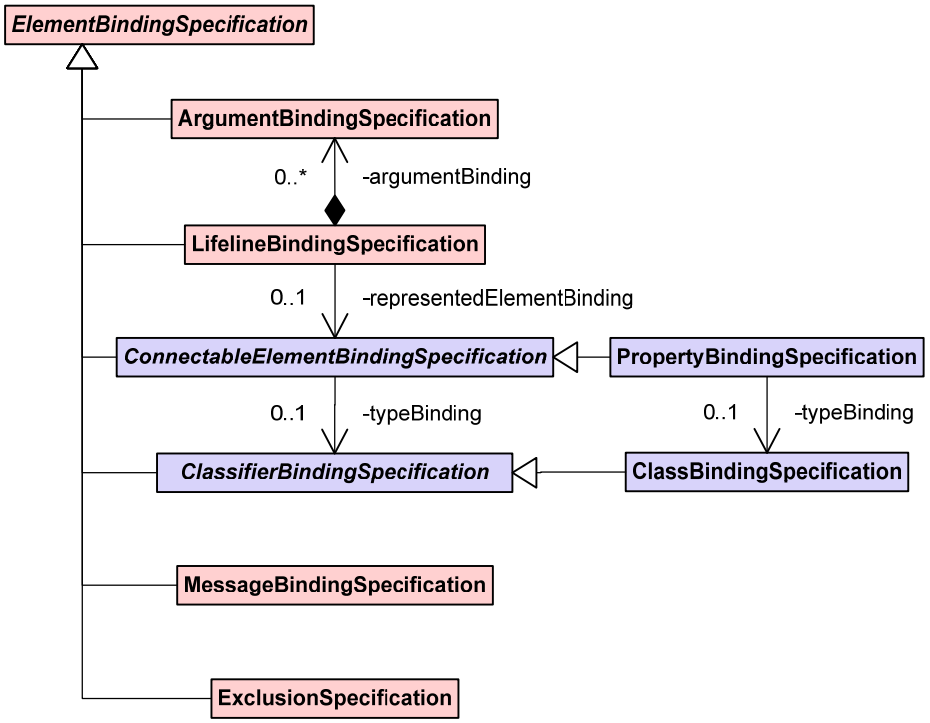

Figure 9-26: Classifier- and ConnectableElementBindingSpecification

9.5 WELL-FORMEDNESS CONSTRAINTS

While some of the well-formedness constraints for AOMDF models are already visually available through the figures in the previous section and partially discussed in the textual descriptions there, we supplement here with a set of constraints expressed formally using OCL [37] in order to rule out more or less obvious illegal models. In our conceptual modeling toolkit for AOMDF, these constraints can be used, for example by the modeling tool component and the model weaver, to perform model checking on both input and output models to enforce and ensure their well-formedness.

We focus our efforts on describing only the most vital invariants for the new concepts introduced by the *AOMDFMM::Interactions* subpackage in our metamodel.

9.5.1 Constraints on Aspect Interactions

Well-formedness of an *AspectInteraction* requires enforcement of the following rules:

- *AspectInteraction*s cannot contain *AspectWeavingPointFragment*s.
- *AspectInteraction*s must contain a non-zero set of *LifelineRole*s.
- If an *AspectInteraction* is of type *simpleAspect* it cannot contain any *AspectAdviceFragment*s, or any *MessageRole*s. It must however contain exactly one *LifelineRole*.
- An *AspectInteraction* of type *compositeAspect* can only contain *AspectAdviceFragment*s directly in its *fragment* set, and maximum one *AspectAdviceFragment* per *AdviceOperator* is allowed.
- If the *AspectInteraction* contains a *MessageRole*, this must be embraced by a *AspectAdviceFragment* with operator *body*.

These rules are stated formally as four OCL invariants in Table 9-2.

Constraints on Aspect Interactions

```
context AspectInteraction

inv: self.fragment
      ->forAll(not oclIsKindOf(AspectWeavingPointFragment))

inv: self.lifeline
      ->select(oclIsKindOf(LifelineRole))->size() > 0


inv: self.aspectInteractionType =
          AspectInteractionType::simpleAspect
      implies
          self.fragment
            ->forAll(not oclIsKindOf(AspectAdviceFragment))
      and self.lifeline
            ->select(oclIsKindOf(LifelineRole))->size() = 1
      and self.message
            ->select(oclIsKindOf(MessageRole))->size() = 0


inv: self.aspectInteractionType =
          AspectInteractionType::compositeAspect
      implies
          self.fragment
            ->forAll(oclIsKindOf(AspectAdviceFragment))
      and self.fragment->size() <= 5
      and self.fragment->size() >= 1
      and self.fragment
            ->select(a : AspectAdviceFragment |
                  a.adviceOperator =  AdviceOperator::begin)
            ->size() < 2
      and self.fragment
            ->select(a : AspectAdviceFragment |
                  a.adviceOperator =  AdviceOperator::before)
            ->size() < 2
```

```
      and self.fragment
            ->select(a : AspectAdviceFragment |
                  a.adviceOperator =  AdviceOperator::body)
            ->size() < 2
      and self.fragment
            ->select(a : AspectAdviceFragment |
                  a.adviceOperator =  AdviceOperator::after)
            ->size() < 2
      and self.fragment
            ->select(a : AspectAdviceFragment |
                  a.adviceOperator =  AdviceOperator::end)
            ->size() < 2
      and self.message
            ->select(oclIsKindOf(MessageRole))->size() > 0
          implies
            ->select(a : AspectAdviceFragment |
                  a.adviceOperator =  AdviceOperator::body)
            ->size() = 1
```

Table 9-2: Constraints on Aspect Interactions

9.5.2 Constraints on Aspect Advice Fragments

The *AspectAdviceFragment*s, used in *AspectInteraction*s of type *compositeAspect*, require well-formedness rules of their own. These are as follows:

- *AspectAdviceFragment*s can only contain one *InteractionOperand*.
- *AspectAdviceFragment*s can not be nested into each other, i.e. the single contained *InteractionOperand* cannot contain an *AspectAdviceFragment*.
- *AspectAdviceFragment*s can not embrace *InteractionFragments* of the types *AspectWeavingPointFragment* and *BehaviourExecution-Specification*. Only *MessageOccurenceSpecification*s and *Standard-CombinedFragments* are allowed.
- *AspectAdviceFragment*s defined with the operator *body* can only embrace *MessageOccurenceSpecification*s that are connected to a *MessageRole*, i.e. they may not embrace *MessageOccurence-Specifications* that connect to ordinary *Message*s. Nor can they embrace *StandardCombinedFragments*. Futhermore, the *body* advice fragments are the only advice fragments where *MessageRoles* are allowed.
- *AspectAdviceFragment*s defined with operators begin, before, after and end may embrace *MessageOccurence-Specifications* that connect to ordinary *Message*s as well as *StandardCombinedFragments*.

We have stated these rules as six invariants in Table 9-3.

Constraints on Aspect Advice Fragments

```
context AspectAdviceFragment

inv: self.operand->size() = 1

inv: self.operand.fragment
      ->forAll(not oclIsKindOf(AspectAdviceFragment))

inv: self.operand.fragment
      ->forAll(not oclIsKindOf(AspectWeavingPointFragment))

inv: self.operand.fragment
      ->forAll(
            not oclIsKindOf(BehaviourExecutionSpecification))

inv: self.adviceOperator <> AdviceOperator::body
      implies
          self.operand.fragment
            ->forAll(m : MessageOccurenceSpecification |
                  not m.message.oclIsKindOf(MessageRole))

inv: self.adviceOperator = AdviceOperator::body
      implies
          self.operand.fragment
            ->forAll(
                  oclIsKindOf(MessageOccurenceSpecification))
      and self.operand.fragment
            ->forAll(m : MessageOccurenceSpecification |
                  m.message.oclIsKindOf(MessageRole))
```

Table 9-3: Constraints on Aspect Advice Fragments

9.5.3 Constraints on Weaving Point Fragments

Requiring enforcement of the following rules will ensure well-formed *AspectWeavingPointFragments* in the primary models:

- An *AspectWeavingPointFragment* can only contain one *InteractionOperand*.
- The number of *LifelineBinding*s contained by an *AspectWeavingPointFragment* must be equal to the number of *LifelineRole*s in the referenced *AspectInteraction*.
- The number of *Lifeline*s covered by an *AspectWeavingPointFragment* must be at greater than or equal to the number of *LifelineRole*s in the referenced *AspectInteraction*.
- The *AspectInteraction* referenced by a *SimpleWeavingFragment* must be of type *simpleAspect*. Likewise the *AspectInteraction* referenced by a *CompositeWeavingFragment* must be of type *compositeAspect*.
- A *SimpleWeavingFragment* can only cover a single *Lifeline,* and its *InteractionOperand* can only contain an empty *fragment* set.

- A *CompositeWeavingFragment* cannot embrace *AspectAdviceFragments*. There are however put no restrictions on the nesting of *AspectWeavingPointFragments*.

- If a *CompositeWeavingFragment* contains a non-zero set of *MessageBindingSpecifications*, the referenced *AspectInteraction* must contain an *AspectAdviceFragment* defined with advice operator *body*.

These rules are expressed as OCL invariants in Table 9-4.

Constraints on Aspect Weaving Point Fragments

```
context AspectWeavingPointFragment

inv: self.operand->size() = 1

inv: self.lifelineBinding->size() =
      self.aspectInteraction.lifeline
            ->select(oclIsKindOf(LifelineRole))->size()

inv: self.coveredLifeline->size() >=
      self.aspectInteraction.lifeline
            ->select(oclIsKindOf(LifelineRole))->size()


-- Constraints for subtype SimpleWeavingFragment --

context SimpleWeavingFragment

inv: self.aspectInteraction.aspectInteractionType =
      AspectInteractionType::simpleAspect

inv: self.coveredLifeline->size() = 1

inv: self.operand.fragment->size() = 0


-- Constraints for subtype CompositeWeavingFragment --

context CompositeWeavingFragment

inv: self.aspectInteraction.aspectInteractionType =
      AspectInteractionType::compositeAspect

inv: self.operand.fragment
      ->forAll(not oclIsKindOf(AspectAdviceFragment))

inv: self.messageBinding->size() > 0
      implies
          self.aspectInteraction.fragment
            ->exists(f : InteractionFragment |
                  f.oclIsKindOf(AspectAdviceFragment)
            and
                  f.oclAsType(AspectAdviceFragment)
                  .adviceOperator = AdviceOperator::body)
```

Table 9-4: Constraints on Weaving Point Fragments

9.6 SUMMARY

In this chapter, we have thoroughly presented the metamodel capturing the abstract syntax required for expressing AOMDF interaction models and weaving instructions. The metamodel has been developed according to the requirements and devised solution strategy presented in the previous chapter.

We have constructed the metamodel as an extension to a simplified UML2 metamodel, and lent some concepts from a simplified metamodel of the UML-based role- and pattern specification language RBML. All new modeling concepts are introduced as well-formed, orthogonal extensions to UML2 and RBML. The concrete syntax for the new modeling concepts has been slightly optimized since the previous proposals which were based on an immature, and partially non-existing, abstract syntax.

Our metamodel forms a solid foundation upon which one can build the model weaver for interaction models, and eventually a complete modeling toolkit for AOMDF.

10 Model Weaver for AOMDF Interaction Models

In the previous chapter we described the metamodel we have developed to enable modeling and weaving of interaction models in AOMDF. The second step of our proof-of-concept work was to develop a model weaver that weaves primary and aspect models conforming to our metamodel. We devised a solution strategy for this in chapter 8.2 where we defined the *weave*-transformation and outlined a design approach using an aspect-oriented, meta-feature injection technique based on Kermeta. In this chapter we proceed with describing the design and implementation results of the *weave*-transformation, as planned in our solution strategy.

10.1 DESIGN

As we described in chapter 8.2, we aimed to build our model weaver in the form of an endogenous, horizontal transformation that takes a tagged primary model and a set of aspect models as input and outputs a composed model. We repeat the earlier presented definition of its entry-point function below.

The weave transformation

```
Composed Model =
      weave( Tagged Primary Model, Set<Aspect Model> )
```

Table 10-1: The weave transformation (once again)

Taking into consideration the requirements and foreseen challenges, we selected to utilize Kermeta to extend our metamodel into an executable metamodel by injecting meta-operations and auxiliary structure into it and modularizing these injections in an aspect oriented fashion (as shown in Figure 8-8). Thereby we could separate lower-level concerns of the weave-transformation from the higher-level logic for weaving of the primary and aspect interactions (as showed in Figure 8-9).

In the subsequent subchapters we present our analysis and design of the high-level weaving logic, the identified lower-level concerns and an overview of the transformation architecture. The implementation is described in chapter 10.2.

10.1.1 High-Level Weaving Logic

The high-level weaving logic is mainly available from the concrete syntax descriptions and conceptual examples in chapter 5.3 and from the AOMDF abstract syntax presentation in [4]. However, these examples and descriptions may not provide a complete understanding of how the weaving of the ordered sets of interaction fragments are performed in general. Therefore, we set up a semi-formal algorithm description of the two weaving kinds in terms of sequences and subsequences of *InteractionFragments*.

We recall from our metamodel that an Interaction contains an ordered set of *InteractionFragment*s which are either atomic, like *MessageOccurenceSpecification*s, or composite like *CombinedFragment*s which in turn contain ordered sets of *InteractionFragment*s in their *InteractionOperand*s (see metamodel extract in Figure 10-1).

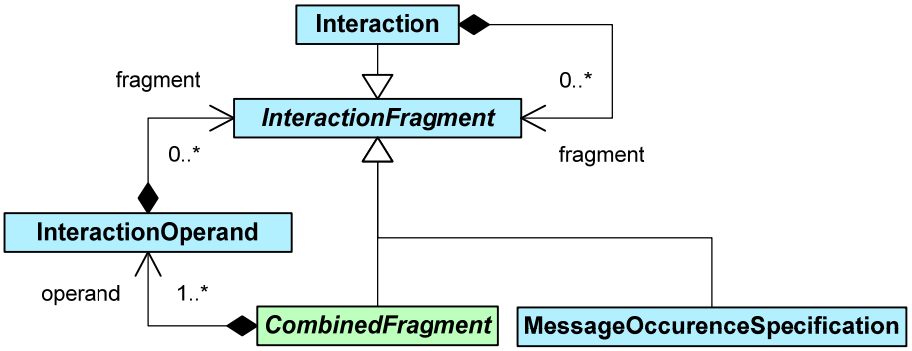

Figure 10-1: Sequentially ordering atomic and composite interaction fragments

It should here be noted that AOMDF defines weaving in terms of interlacing and merging of *Message*s. However, since the ordering of *MessageOccurenceSpecification*s (i.e. the *MessageEnd*s) in *Interaction*s or *InteractionOperand*s is what governs the order in which the *Message*s occur in an interaction, this is what the weaver should initially weave. After weaving the *InteractionFragments*, any *Message*s can be copied or merged as necessary into the *message*-set of the target *Interaction*. In other words the ordering of the *message*-set contained by an *Interaction* is irrelevant for retaining the time sequence for the *Message*s.

10.1.1.1 Weaving of simpleAspect

In the *simpleAspect* situation we start with a primary sequence $P = P_{1..n}$, into which we insert a *SimpleWeavingFragment* at index *k*. The primary sequence then becomes a tagged primary sequence:

$$TP = P_{1..(k-1)}, w^{simple}, P_{(k+1)..n}$$

where w^{simple} is a *SimpleWeavingFragment* that partitions the original sequence and references the simpleAspect:

$$A^{simple} = A_{1..m}.$$

Upon weaving, the entire aspect sequence is inserted into the primary sequence at the index of the weaving point. The resulting output is then the interaction sequence:

$$C = P_{1..(k-1)}, A_{1..m}, P_{(k+1)..n}.$$

10.1.1.2 Weaving of compositeAspect

In the *compositeAspect* case, the primary sequence $P = P_{1..n}$ is tagged with a *CompositeWeavingFragment* covering the subsequence of interaction elements from indexes *k* to *m* over which the *compositeAspect* is to be adapted. The *CompositeWeavingFragment* will then in the tagged sequence replace that subsequence and rather embrace it in its *InteractionOperand*. We express this as follows:

$$TP = P_{1..(k-1)}, w^{composite}, P_{(m+1)..n} \quad \wedge \quad w^{composite} = \left\langle P_{k..m} \right\rangle.$$

The *compositeAspect* referenced by the *CompositeWeavingFragment* can be viewed as a multisequence containing a sequence of interaction elements for each of the *AdviceOperator*s:

$$A^{composite} = \left\langle A_{1..v}^{begin} \right\rangle, \left\langle A_{1..w}^{before} \right\rangle, \left\langle A_{1..x}^{body} \right\rangle, \left\langle A_{1..y}^{after} \right\rangle, \left\langle A_{1..z}^{end} \right\rangle$$

The combining of subsequences in order to obtain the woven sequence is done according to the following rule:

$$\begin{aligned} C = P_{1..(k-1),}\ & A_{1..v}^{begin}, \\ & A_{1..w}^{before}, \left(p_k \circ a_1^{body}\right), \ldots, \left(p_k \circ a_x^{body}\right), A_{1..y}^{after}, \\ & \vdots \\ & A_{1..w}^{before}, \left(p_m \circ a_1^{body}\right), \ldots, \left(p_m \circ a_x^{body}\right), A_{1..y}^{after}, \\ & A_{1..z}^{end}, P_{(m+1)..n} \end{aligned}$$

where ***(x*** $\circ$ ***y)*** denotes the atomic weaving (i.e. merging) of two elements.

In Kermeta we can employ the natively available, generic data structures *OrderedSet* or *Sequence* to easily extract, recombine and otherwise manipulate sequences of model elements.

10.1.1.3 Lifelines and underlying structure

In both the *simpleAspect*- and *compositeAspect*-case, weaving of *Lifeline*s and their underlying structural elements (*ConnectableElement* and *Classifier*) is also an important part of the high-level weaving logic; however, this is

trivially understood from previous examples and the presentation of our metamodel, so we do not repeat this here.

10.1.2 Low-Level Concerns

Based on the pseudo-algorithm presented in chapter 8.2.1.1 and the weaving logic presented in the previous section, we identify the following six low-level concerns in our *weave*-transformation.

10.1.2.1 Navigation

Any injected meta-features supporting easier navigation of models should be organized as an isolated concern.

10.1.2.2 Deep-Copy

Like many other object-oriented languages, Kermeta provides a clone operation for all objects. However this operation only performs a deep-copy of the object being cloned and its attributes, while any references to other objects are, naturally, shallow-copied. This imposes certain difficulties when copying model elements from the source space to the target space in the transformation. Thus, we need to implement some operations that may allow us to simply deep-copy a model or a certain model segment (e.g. an interaction and its contained lifelines, messages and fragments) on the fly with a single operation-call.

A sub-concern of this is the traceability needed to reconstruct references properly when an element is copied from source to target space. Operation and features related to this sub-concern do not need to be separately organized and can be implemented with the deep-copy concern.

10.1.2.3 Signature Comparison

Comparison of signatures for model elements representing the underlying structure model for an interaction is required for avoiding collisions during weaving. We identify this as an isolatable concern.

10.1.2.4 Atomic Weaving

Another separable concern is the atomic weaving of model elements, i.e. weaving of a certain UML2 concept with its role-variant from RBML (for example a *Lifeline* and a *LifelineRole*).

10.1.2.5 Extraction of Weaving Points

The extraction of weaving instructions from tagged primary models, including the element binding information captured within these, is yet another concern of its own.

10.1.2.6 Extraction of Aspect Advice

Finally, advice extraction from aspect interactions (*simpleAspect*-type as well as *compositeAspect*-type) is also a separable concern.

The meta-operations and auxiliary meta-features that we inject into our metamodel to make it executable can, in addition to the class-encapsulation dimension, be modularized as aspects according to the six above concerns.

10.1.3 Transformation Organization

In line with our solution strategy, we organize the transformation code in layers with low-level concerns at the bottom, and in that way form an executable metamodel which complex transformations can be programmed against at a higher level of abstraction. The low-level concerns are grouped into aspects. Some of them have dependencies to others. We illustrate the organization of the transformation in Figure 10-2.

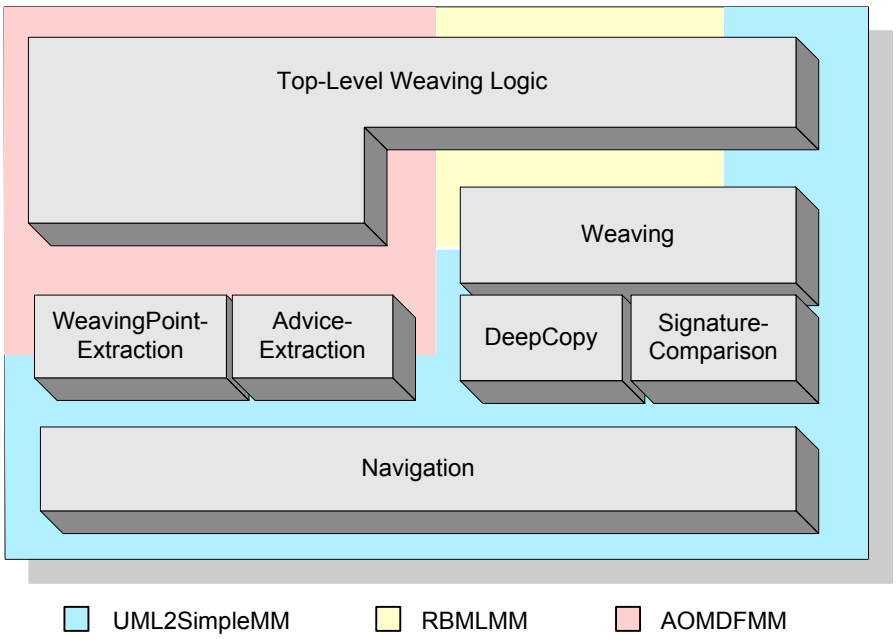

Figure 10-2: Transformation concerns crosscutting the abstract syntax

As shown, the aspects *deep-copy*, *signature comparison*, *advice extraction* and *weavingpoint extraction* all depend on the *navigation* aspect. *Atomic weaving* is dependant on the *deep-copy* and *signature comparison* aspects. The top-level weaving transformation logic depends on *weavingpoint extraction*, *advice extraction*, and *atomic weaving*. The background colors of Figure 10-2 show how the concerns crosscut the abstract syntax in our metamodel.

10.2 IMPLEMENTATION

The implementation is done by loading our metamodel into Kermeta. This gives access to treat all the metaclasses as native Kermeta classes. We then implement the low-level transformation aspects, before finally implementing the top-level weaving logic.

We describe only selected key parts of our implementation here. The complete source code for each of the transformation aspects (and the top-level weaving logic) is too large for inclusion in this document. Readers interested in obtaining the complete source code artifacts should refer to appendix A.

Kermeta syntax is pretty much like OCL and working with Collections in Java. We therefore find it unnecessary to describe any essential code listings in detail through the remainder of this chapter.

10.2.1 Package Structure

Extending the package namespaces used for our metamodel we introduce our model weaver as a new subpackage in the *AOMDFMM* package, and name it *InteractionWeaver* as shown in Figure 10-3. *InteractionWeaver* imports the *Models* and *Interactions* subpackages, and thereby has access to all concepts in our metamodel.

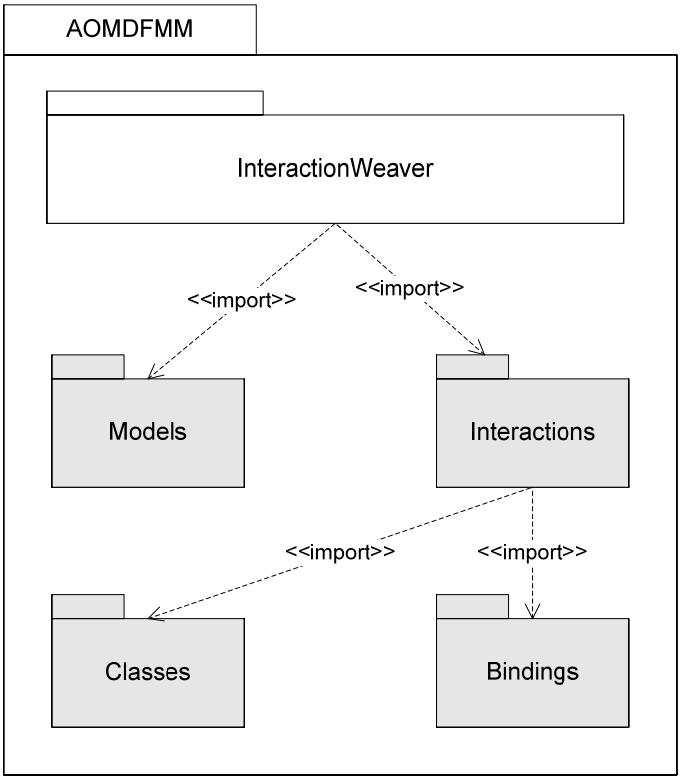

Figure 10-3: Introducing InteractionWeaver as a subpackage in AOMDFMM

We further divide the *InteractionWeaver* package into the two subpackages *TransformationAspects* and *Util* as shown in Figure 10-4. The first subpackage contains the implementations of the lower-level transformation aspects, while the latter contains utility classes used in the top-level transformation. The classes implementing the top-level transformation logic are contained directly in the *InteractionWeaver*-package. We present this and the contents of the two subpackages in the next subchapters.

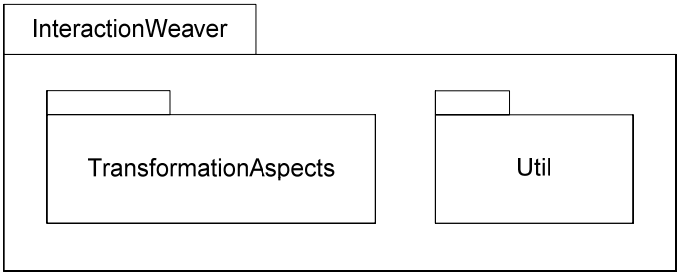

Figure 10-4: Subpackages of InteractionWeaver

10.3 TRANSFORMATION ASPECTS

In Figure 10-5 we show the contents of the *TransformationAspects* package in the form of a package diagram with one subpackage per low-level concern shown in Figure 10-2 (compare the imports to dependencies described earlier). Note however that these subpackages do not exist as separate namespaces in the dimension of earlier described packages. Recall from Figure 10-2, that these aspects crosscut the metamodel, and hence also the metamodel packages. (For the sake of being able to tell them apart from the ordinary packages, we depict their membership in a different dimension by adding an icon to the package symbols).

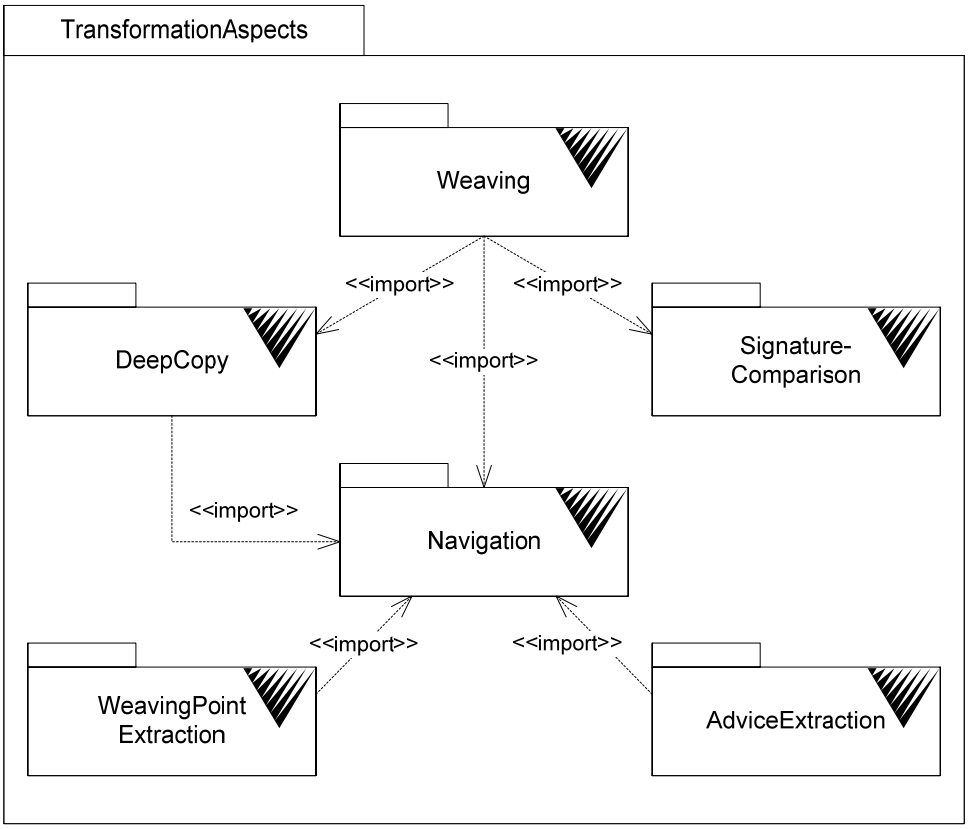

Figure 10-5: Aspects in TransformationAspects package

10.3.1 Navigation

A major challenge in manipulating the sequences of interaction fragments during weaving, and other lower-level concerns, is the complexity caused by the fact that in the metamodel (and in UML2), both *Interaction* and *InteractionOperand* may contain sequences of *InteractionFragment*s. Although not a great challenge in terms of pure navigation, this forces the duplication of any code that manipulates the fragment sequences into both the *Interaction* and *InteractionOperand* metaclasses, and hence makes it difficult to write recursive or pseudo-recursive operations in various situations where suitable.

We can overcome this complexity by introducing an auxiliary common supertype for both *Interaction* and *InteractionOperand* as shown in Figure 10-6. Table 10-2 shows the trivial Kermeta code required to accomplish this. The *InteractionFragmentContainer* metaclass is defined as a new class inheriting from Element (the root node in UML2 and our metamodel). We enforce inheritance of this new class onto *Interaction* and *InteractionOperand* by reopening their class definitions using the "@aspect"-annotation in Kermeta and defining new features to be injected into the existing definitions.

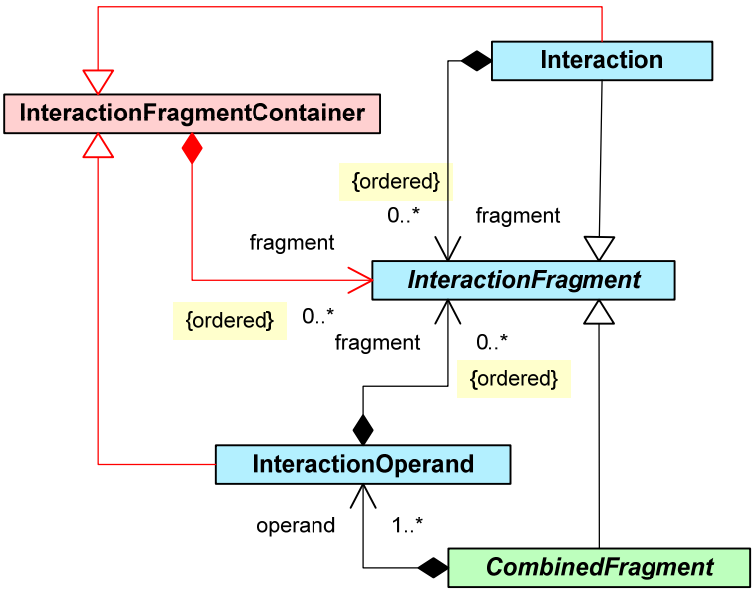

Figure 10-6: Introducing auxiliary metaclass InteractionFragmentContainer

TransformationAspects::Navigation

```
package AOMDF::UML2SimpleMM::Interactions;
...
class InteractionFragmentContainer inherits Element
{
      attribute fragment : InteractionFragment[0..*]
}

@aspect "true"
class Interaction inherits InteractionFragmentContainer
{
}
```

```
@aspect "true"
class InteractionOperand
      inherits InteractionFragmentContainer
{
}
```

Table 10-2: Injecting auxiliary features for the navigation-aspect

10.3.2 DeepCopy

In Table 10-3 we list an extract of code injection for the deep-copy aspect. We instrument an operation *deepCopy(src : Element)* into the root-level metaclass of *UML2SimpleMM*, *Element*, and override this throughout the hierarchy of metaclasses. This is analogous to a copy-constructor which takes the object to create a copy of as argument.

In *Element* we also instrument a reference to a *target Element* which is set upon call to *deepCopy*. This enables the model elements in the source space to know about there corresponding copy in the target space during a transformation execution. This enables us to implement operations of the form *resolve*X*()* where X is some reference that needs to be resolved at post-copy-time and thus we can do deep-copying of model elements and properly reconstruct the references between them in the target space.

TransformationAspects::DeepCopy (1)

```
@aspect "true"
class Element
{
      reference target : Element

      operation deepCopy( src : Element )
      is do
            src.target := self
      end
}
...
@aspect "true"
class StructuralFeature
{
      method deepCopy( src : StructuralFeature )
            from NamedElement
      is do
            super( src )
            ...
      end
}
...
@aspect "true"
class Class
{
      method deepCopy( src : Class )
      is do
            super( src )
            ...
      end
```

```
      operation resolveTargetSuperClass()
      ...
}
```

Table 10-3: Extract from DeepCopy (1)

Likewise, we can now utilize the *InteractionFragmentContainer* auxiliary class to deep-copy *InteractionFragments* as shown in Table 10-4. Additionally, we have also implemented operations for deep-copying of entire primary and tagged models.

TransformationAspects::DeepCopy (2)

```
alias MsgOcc : …::MessageOccurenceSpecification;
alias BhvOcc : …::BehaviourOccurenceSpecification;
alias StandardCF : …::StandardCombinedFragment;

@aspect "true"
class InteractionFragmentContainer
{
      operation deepCopyFragments(
            src : InteractionFragmentContainer )
      is do
        src.fragment.each{ srcIf |
            if srcIf.isKindOf( MsgOcc ) then
              var tgtIf : MsgOcc init MsgOcc.new
              tgtIf.deepCopy(srcIf.asType(MsgOcc))
              self.fragment.add( tgtIf )
            else if srcIf.isKindOf( BhvOcc ) then
              var tgtIf : BhvOcc init BhvOcc.new
              tgtIf.deepCopy(srcIf.asType(BhvOcc))
              self.fragment.add( tgtIf )
            else if srcIf.isKindOf( StandardCF ) then
              var tgtIf : StandardCF init StandardCF.new
              tgtIf.deepCopy(srcIf.asType(StandardCF))
              self.fragment.add( tgtIf )
            end
            end
            end
        }
      end

      operation resolveTargetFragments()
      is do
            ...
      end
}
```

Table 10-4: Extract from DeepCopy (2)

10.3.3 Signature Comparison

Table 10-5 shows an extract from the implementation of the signature comparison aspect, which like deep-copying exploits the existing class

hierarchy by letting subclasses of Element implement the abstract operation *sigEquals(e : Element)*.

TransformationAspects::SignatureComparison

```
@aspect "true"
class Element
{
      operation sigEquals( e : Element ) : Boolean
      is abstract
}
...
@aspect "true"
class NamedElement
{
      method sigEquals( e : NamedElement ) : Boolean
      is do
            ...
      end
}
...
@aspect "true"
class Operation
{
      method sigEquals( e : Operation ) : Boolean
            from NamedElement
      is do
            result := super( e )
              and parameter.size() == e.parameter.size()

            if result == true then
              from var i : Integer init 0
              until i == parameter.size() or result == false
              loop
                  result :=
                        parameter.elementAt(i).sigEquals(
                              e.parameter.elementAt(i) )
              end
            end
      end
}
```

Table 10-5: Extract from the SignatureComparison

10.3.4 Weaving

The atomic weaving operations are also organized like deep-copying and signature-comparison in order to exploit the class hierarchy. This is shown in Table 10-6.

We also organize some of the interaction sequence manipulation operations into this aspect and encapsulate them in our auxiliary class *InteractionFragmentContainer*. Table 10-7 shows the *injectFragments* operation which is used by the top-level weaver logic to inject a sequence of interaction fragments at some point in either an InteractionOperand or an Interaction. Compare the code in Table 10-7 with the sequence manipulations described in section 10.1.1.

TransformationAspects::Weaving (1)

```
@aspect "true"
class Element
{
      operation weave( role : ElementRole )
      is do
            role.target := self
      end
}
...
...
@aspect "true"
class Class
{
   method weave( role : ClassRole )
   is do
   ...
   ...
   super( role )
   end
}
...
...
@aspect "true"
class Lifeline
{
   method weave( role : LifelineRole ) is
   do
      super( role )
   end
}
```

Table 10-6: Extract from Weaving (1)

TransformationAspects::Weaving (2)

```
@aspect "true"
class InteractionFragmentContainer
{
      operation injectFragments( injectableFragments :
            OrderedSet<InteractionFragment>,
            preceding : InteractionFragment,
            succeeding : InteractionFragment )
      is do
            var precedingFragments :
                  OrderedSet<InteractionFragment>
            var succeedingFragments :
                  OrderedSet<InteractionFragment>

            if not preceding.isVoid then
                  precedingFragments := fragment.subSet(0,
                        fragment.indexOf(preceding))
            end

            if not succeeding.isVoid then
                  succeedingFragments :=
                  fragment.subSet(
                        fragment.indexOf(succeeding),
```

```
                        fragment.size)
            end

            fragment.clear()

            if not precedingFragments.isVoid then
                  fragment.addAll( precedingFragments )
            end

            fragment.addAll( injectableFragments )

            if not succeedingFragments.isVoid then
                  fragment.addAll( succeedingFragments )
            end
      end
}
```

Table 10-7: Extract from Weaving (2)

10.3.5 WeavingPointExtraction

Table 10-8 and Table 10-9 together show the complete code for extracting weaving points from a tagged primary model's interactions.

TransformationAspects::WeavingPointExtraction (1)

```
package AOMDF::AOMDFMM::Models;
...
@aspect "true"
class TaggedPrimaryModel
{
      operation getAllWeavingPoints() :
            AspectWeavingPointFragment[0..*]
      is do
            result :=
              OrderedSet<AspectWeavingPointFragment>.new
            result.addAll( self.getWeavingPoints() )
      end
}


package AOMDF::UML2SimpleMM::Classes;
...
@aspect "true"
class Package
{
      operation getWeavingPoints() :
            AspectWeavingPointFragment[0..*]
      is do
            result :=
              OrderedSet<AspectWeavingPointFragment>.new
              self.classifier.each{ cls |
                  if cls.isKindOf(Interaction) then
                    var i : Interaction init
                    cls.asType(Interaction)
                    result.addAll( i.getWeavingPoints() )
                  end
              }
```

```
                self.subPackage.each{ subPck |
                    result.addAll( subPck.getWeavingPoints() )
                }
        end
}
```

Table 10-8: Extract from WeavingPointExtraction (1)

TransformationAspects::WeavingPointExtraction (2)

```
package AOMDF::UML2SimpleMM::Interactions;
...
@aspect "true"
class InteractionFragmentContainer
{
        operation getWeavingPoints() :
                AspectWeavingPointFragment[0..*]
        is do
                result :=
                  OrderedSet<AspectWeavingPointFragment>.new
                  self.fragment.each{ f |
                    if f.isKindOf( SimpleWeavingFragment ) then
                      result.add(
                      f.asType(SimpleWeavingFragment))
                    else if f.isKindOf(
                            CompositeWeavingFragment ) then
                      result.add(
                            f.asType(CompositeWeavingFragment) )
                    end
                    end

                    if f.isInstanceOf( CombinedFragment ) then
                      f.asType(CombinedFragment).operand
                      .each{ op | result.addAll(
                            op.getWeavingPoints() ) }
                    end
                }
        end
}
```

Table 10-9: Extract from WeavingPointExtraction (2)

10.3.6 AdviceExtraction

Extraction of interaction fragments from an aspect interaction of type *simpleAspect* is trivial. For the *compositeAspect*, the complete code for easy extraction of the interaction sequences inside the advice fragments is shown in Table 10-10.

TransformationAspects::AdviceExtraction

```
package AOMDF::AOMDFMM::Interactions;
...
@aspect "true"
class AspectInteraction
{
      operation getAdviceBegin() :
            InteractionFragmentContainer
      is do
            result := getAdvice( AdviceOperator.begin )
      end

      operation getAdviceBefore() :
            InteractionFragmentContainer
      is do
            result := getAdvice( AdviceOperator.before )
      end

      operation getAdviceBody() :
            InteractionFragmentContainer
      is do
            result := getAdvice( AdviceOperator.body )
      end

      operation getAdviceAfter() :
            InteractionFragmentContainer
      is do
            result := getAdvice( AdviceOperator.after )
      end

      operation getAdviceEnd() : InteractionFragmentContainer
      is do
            result := getAdvice( AdviceOperator.~end )
      end

      operation getAdvice( operator : AdviceOperator ) :
            InteractionFragmentContainer
      is do
            result := void

            from var i : Integer init 0
            until i == self.fragment.size and result.isVoid
            loop
              var f : InteractionFragment
                  init self.fragment.elementAt(i)

              if f.isKindOf( AspectAdviceFragment ) then
                  var af : AspectAdviceFragment init
                        f.asType(AspectAdviceFragment)

                  if af.adviceOperator == operator then
                        result := af.operand.first
                  end
              end
              i := i + 1
            end
      end
}
```

Table 10-10: AdviceExtraction

10.4 UTILITIES

The *Util* subpackage, shown in Figure 10-7, currently contains two generic, utility classes used by the top-level weaver logic to keep track of atomic weavings of model elements during an execution of the *weave*-transform. This way the weaver can avoid unwanted repetition of atomic weavings in cases where for example an aspect interaction is adapted twice into a primary model and the underlying class models only need to be woven once. These utility classes are *OneToManyTrace* and its specialization *Bidirectional-OneToManyTrace*.

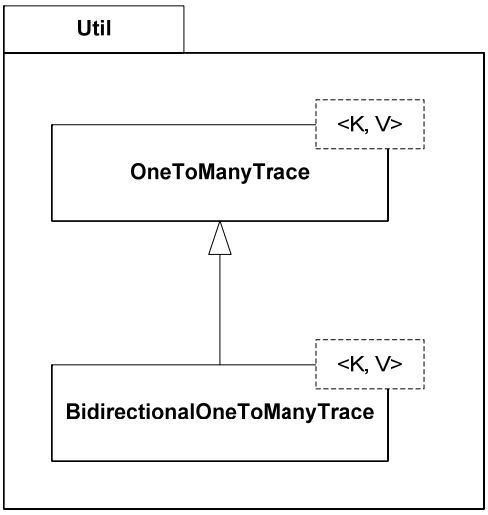

Figure 10-7: Auxiliary, generic classes in the Util package

As the names suggest, *OneToManyTrace* and *BidirectionalOneToManyTrace* provide data-structures (with related operations) for storing unidirectional and bidirectional one-to-many element mappings. The underlying data-structure is a generic *HashBag* based on a *HashTable* with the value-type parameter set to be a generic *OrderdSet*. In Table 10-11 and Table 10-12 we briefly list the available operations in these two utility classes.

Util::OneToManyTrace

```
class OneToManyTrace<SRC, TGT>
{
      reference src2tgt : Hashtable<SRC, OrderedSet<TGT>>

      /**
       * Stores a trace. The TGT element is added to
       * the set of values for the key SRC.
       **/
      operation storeTrace(src : SRC, tgt : TGT)
      …

      /**
       * Returns Set of TGT elements for a given SRC element.
       **/
      operation getTargetElems( src : SRC ) : OrderedSet<TGT>
      …
```

```
        /**
         * Returns Set of all SRC elements stored in the Trace.
         **/
        operation getAllSourceElems() : Set<SRC>
        …
}
```

Table 10-11: Operations of OneToManyTrace

Util::BidirectionalOneToManyTrace

```
class BidirectionalOneToManyTrace<SRC, TGT>
        inherits OneToManyTrace<SRC, TGT>
{
        reference tgt2src : Hashtable<TGT, SRC>
        …
        …

        /**
         * Returns SRC element for a given TGT element.
         */
        operation getSourceElem(tgt : TGT) : SRC
        …

        /**
         * Returns Set of all TGT elements stored in the Trace.
         **/
        operation getAllTargetElems() : Set<TGT>
        …
}
```

Table 10-12: Operations of BidirectionalOneToManyTrace

10.5 TOP-LEVEL WEAVER LOGIC

The low-level transformations aspects and the metamodel together constitute an executable metamodel which allows us to program complex, heavy-duty transformations at a higher level of abstraction. This is clearly visible in Table 10-13 where we show the top-level operation *weave*, i.e. the entry point of the model weaver. The code is described with inline comments.

The weave-transformation

```
operation weave( taggedModel : TaggedPrimaryModel,
   aspectModels : AspectModel[0..*] ) : ComposedModel
is do
   ...
   ...
   ...
   //instantiate the resulting model
   result := ComposedModel.new

   //copy the primary model part from the tagged model
   var tm : TaggedPrimaryModel init TaggedPrimaryModel.new
   tm.deepCopy( taggedModel )
   result.addPackages( tm.subPackage )
```

```
   //copy aspect models (class part only)
   aspectModels.each{ srcAm |
      var am : AspectModel init AspectModel.new
      am.deepCopy( srcAm )
      result.addPackages( am.subPackage )
   }

   //give some meaningful name to output model
   result.name := taggedModel.name + "_ComposedWith"
   aspectModels.each{ srcAm |
      result.name.append("_" + srcAm.name)
   }

   //fetch all weavingpoints from source tagged model
   var wpoints : OrderedSet<AspectWeavingPointFragment>
      init taggedModel.getAllWeavingPoints()

   wpoints.each{ wp |
→     expandLifelineBindings( wp )
→     processWeavingPoint( wp )
   }

   //do some final re-resolving on the Aspect packages
   aspectModels.each{ srcAm |
      result.resolveClassModel( srcAm )
   }

   //post-weaving fix of any colliding names (may potentially
   // result from multiple adaptations of same aspect, etc)
   ensureUniqueTargetNames()
end
```

Table 10-13: Implementation of the weave operation (top-level transformation)

10.5.1 Processing of WeavingPoints

The weave-operation described in Table 10-13 calls the operation *processWeavingPoint* in order to process each weaving point. This operation is listed in Table 10-14 below with inline comments describing the code. Althought the algorithm still may seem a bit complex upon first look, the abstraction provided by the lower-level transformation aspect layers yields an implementation that is is easy to comprehend.

Processing of weaving points

```
operation processWeavingPoint(
   wp : AspectWeavingPointFragment )
is do
   //Locate target Interaction node
   var tgtIA : Interaction init
      wp.getContainingInteraction.target.asType(Interaction)

   //Inject non-role lifelines and properties to Interaction
   wp.aspectInteraction.getNonRoleLifelines().each{ srcLL |
      var srcPP : Property init
         srcLL.represents.asType(Property)
```

```
   if not tgtIA.hasProperty( srcPP )
   then
      var tgtPP : Property init Property.new
      tgtPP.deepCopy( srcPP )
      srcPP.resolveTargetType()
      tgtIA.structuralFeature.add( tgtPP )
      asp2tgt_Property.storeTrace( srcPP, tgtPP )
   end

   if not tgtIA.hasLifeline( srcLL ) then
      var tgtLL : Lifeline init Lifeline.new
      tgtLL.deepCopy( srcLL )
      srcLL.resolveTargetRepresents()
      tgtIA.lifeline.add( tgtLL )
      asp2tgt_Lifeline.storeTrace( srcLL, tgtLL )
   end
}

//weave lifeline, properties and classes
//(avoids re-weaving)
wp.lifelineBinding.each{ lb |

   var pb : PropertyBindingSpecification init
      lb.representedElementBinding
      .asType(PropertyBindingSpecification)

   var cb : ClassBindingSpecification init
      pb.typeBinding.asType(ClassBindingSpecification)

   //weave the types of the properties
   var primaryTgtClass : Class init
      cb.primary.target.asType(Class)

   if role2tgt_Class.getTargetElems(cb.role)
       .excludes( primaryTgtClass ) then
      primaryTgtClass.weave(cb.role)
      role2tgt_Class.storeTrace(cb.role, primaryTgtClass)
   end

  //weave properties represented by lifelines
   var primaryTgtProp : Property init
      pb.primary.target.asType(Property)

   if role2tgt_Property.getTargetElems(pb.role)
      .excludes( primaryTgtProp )
   then
      primaryTgtProp.weave(pb.role)
      role2tgt_Property.storeTrace(pb.role, primaryTgtProp
   end

   //weave lifelines
   var primaryTgtLL : Lifeline init
      lb.primary.target.asType(Lifeline)

   if role2tgt_Lifeline.getTargetElems(lb.role)
     .excludes( primaryTgtLL )
  then
      primaryTgtLL.weave(lb.role)
      role2tgt_Lifeline.storeTrace(lb.role, primaryTgtLL )
   end
```

```
   }

   //copy messages from aspect to target
   wp.aspectInteraction.message.each{ srcMsg |
      var tgtMsg : Message init Message.new
      tgtMsg.deepCopy( srcMsg )
      tgtIA.message.add( tgtMsg )
      asp2tgt_Message.storeTrace( srcMsg, tgtMsg)
   }

   //determine the container, the predecessor and the
   //successor of the weavingpoint
   var tgtPredecessor : InteractionFragment init
   wp.getPrecedingInteractionFragment().target.
      asType(InteractionFragment)
   var tgtSuccessor : InteractionFragment init
   wp.getSucceedingInteractionFragment().target.
      asType(InteractionFragment)

   //now perform weaving / injection of fragments according
   //to the weavingpoint
   var tgtIFC : IFC init wp.getContainer().target.asType(IFC)
   var tmpIFC : InteractionFragmentContainer
      init InteractionFragmentContainer.new

   if wp.isKindOf( SimpleWeavingFragment ) then
      tmpIFC.deepCopyFragments( wp.aspectInteraction )
   else if wp.isKindOf( CompositeWeavingFragment )
   then
→     tmpIFC := processCompositeWeaving(
         wp.asType(CompositeWeavingFragment) )
   end
   end

   tgtIFC.injectFragments( tmpIFC.fragment,tgtPredecessor,
      tgtSuccessor )

   wp.aspectInteraction.resolveTargetFragments()
   wp.aspectInteraction.message.each{ m |
      m.resolveTargetReceiveEvent()
      m.resolveTargetSendEvent()
   }
end
```

Table 10-14: The processWeavingPoint-operation

Note here that *processWeavingPoint* may call the operation *processCompositeWeaving* which we do not show here (refer to appendix A for details on how to obtain the complete source code).

10.5.2 Expansion of Lifelinebindings

Finally, we show in Table 10-15 the operation *expandLifelineBindings* which expands the LifelineBindingSpecifications with a typeBinding and representedElementBinding, and in that way facilitate that the weaving instructions for the interactions drive a weaving of the underlying class models.

Expansion of LifelineBindings

```
operation expandLifelineBindings(
   wp : AspectWeavingPointFragment )
is do
   wp.lifelineBinding.each{ lb |
      var pb : PropertyBindingSpecification init
         PropertyBindingSpecification.new

      var cb : ClassBindingSpecification init
         ClassBindingSpecification.new

      pb.primary := lb.primary.represents.asType(Property)
      pb.role := lb.role.represents.asType(PropertyRole)

      cb.primary := pb.primary.type.asType(Class)
      cb.role := pb.role.type.asType(ClassRole)

      pb.typeBinding := cb
      lb.representedElementBinding := pb
   }
end
```

Table 10-15: Derivation of bindings for underlying structure elements

10.6 SUMMARY

We have in this chapter presented the major points from our design and implementation of a model weaver for AOMDF interaction models. The model weaver has been constructed according to the requirements and solution strategy devised in chapter 8.2, and is built as an endogenous, horizontal transformation.

Due to the heavy-duty nature of this transformation, we have used an approach based on the Kermeta language where we instrument the metamodel developed in the previous chapter with meta-operations and auxiliary meta-features in an aspect oriented fashion. The result is an executable metamodel upon which the transformation logic can be expressed at a higher level of abstraction.

The model weaver has been informally tested and validated during development.

PART IV

DISCUSSION

11 Conclusion

In this chapter we conclude our work with a summary and claim of primary and secondary contributions of this thesis. Known weaknesses of our work are also presented. In the next two chapters we look into related work and future work ideas, respectively.

11.1 SUMMARY

The motivation behind this thesis was the lack of a proof-of-concept for automatic weaving of interaction models (sequence diagrams) in the Aspect Oriented Model Driven Framework (AOMDF). The ideal proof-of-concept would certainly have been a prototype of a complete modeling toolkit for AOMDF, however we limited our scope to develop only two vital parts:

- metamodel capturing the abstract syntax of AOMDF
- model weaver for interaction models conforming to the metamodel

The metamodel was developed as an extension to a simplified UML2 metamodel and a simplified metamodel of RBML (a UML-based language for role- and pattern specification). All new modeling concepts were introduced as orthogonal extensions to UML2 and RBML.

The model weaver was constructed as an endogenous, horizontal transformation. The Kermeta language was used to extend the metamodel with an executable layer for handling low-level transformation concerns. The high-level transformation logic was then programmed against this new layer at an higher level of abstraction. The lower-level transformation concerns were both encapsulated into the metaclasses and modularized as aspects (i.e. multidimensional separation of concerns).

Informal testing and validation of the metamodel and model weaver was conducted.

11.2 WEAKNESSES

In the aftermath we see that both the metamodel and the model weaver could use some further reconsideration on several issues. However, we have always beared in mind that our goal is to do a proof-of-concept on automating weaving of interaction models in AOMDF, and not a final implementation of some tool.

In general, the metamodel can benefit from certain design improvements, such as creating subtypes of *AspectInteraction* instead of using an Enumeration to tell the two kinds apart. For some metaclasses, like the derivable binding specifications, it is questionable whether they should be part of the abstract syntax or moved to the auxiliary layer (coded in Kermeta) instead.

The model weaver contains a handful of minor bugs and errors, and a few shortcomings with respect to proper weaving of advice-fragments with the operator body (including message-weaving). Fixing these is simply a matter of time.

11.3 CLAIMED CONTRIBUTION

We divide the contribution claims of this thesis into primary and secondary.

11.3.1 Primary Contributions

11.3.1.1 Metamodel

The metamodel we have developed provides a solid foundation for tool-support and future work on the behavior modeling area of AOMDF. Its organization and reuse of UML2 and RBML keeps the number of entirely new concepts needed to a minimum, yet facilitating natural extension of AOMDF to cover other areas of UML in the future.

Another positive outcome of the new metamodel is that we have managed to make improvements to previous proposals of concrete syntax for AOMDF interactions. While the previous proposals relied on heavy usage of stereotypes and a UML profile, we are now more in line with the teachings of language driven development and not locked into a language extension mechanism that may not scale as AOMDF expands.

Our metamodel also shows that the abstract syntax for AOMDF interaction models can be constructed as an orthogonal extension to UML2.

11.3.1.2 Model Weaver

The model weaver successfully automates the weaving of tagged primary interactions and aspect interactions, at least from a proof-of-concept perspective. Hence we can conclude that it fulfills our proof-of-concept goal. Further, it provides a validation of the metamodel as fit for purpose.

All challenges encountered during the model weaver implementation break down to two main challenges which we foresaw in our solution approach. One is the complexity of the UML2 metamodel. The second is the large amounts of data contained by even the simplest interaction model.

11.3.2 Secondary Contributions

11.3.2.1 Coupling Class- and Interaction Weaving

In our model weaver, we have chosen to let the interaction models that are subject to weaving dictate a weaving of the underlying class models. We facilitated this in the abstract syntax as derivable binding specifications and showed in our implementation how a lifeline-binding can be expanded to contain a class- and property-binding. This allowed us to weave and connect the static structure underlying the interactions before the interaction weaving began. The relationship between class model weaving and interaction model weaving in AOMDF is something that should be investigated further, and our minor contribution on this may form the basis for such an investigation.

11.3.2.2 Advanced Separation of Concerns in Model Transformations

In short, the overall approach employed by us in this thesis has been to first develop a metamodel, then make it executable and finally build an advanced transformation on top of it. By letting the executable layer of the metamodel serve as an API between the metamodel and the transformation, this allows the transformation code to be expressed at a higher level of abstraction.

While there is nothing new to this approach, the choice of using Kermeta has enabled us to exercise separation of concerns one step further in a lightweight, aspect oriented fashion while developing the executable layer. We hope that this thesis can function as a “guide-by-example” for effectively modularizing heavy-duty transformations.

12 Related Research

Model composition or model merging are popular topics in recent research and have been investigated by several.

A significant work on weaving interaction models is that of Klein et al. [57] who propose a technique to statically weave behavioral aspects into sequence diagrams using an automated process that takes into account the semantics of the models. The algorithms used are proposed as a formally proved merge operator, with an accompanying implementation in Kermeta.

13 Future Work

In this chapter we outline ideas and plans for future work.

13.1 SOLIDIFY CURRENT WORK

From a narrow perspective, future work should consist of solidification of the work presented in this thesis. Investigating how to combine the earlier proof-of-concept on class-weaving with our proof-of-concept on interaction-weaving will put the metamodel to a test and is a suitable starting point. Some of this work has already began and is described in [43].

Further we should seek to formalize the weaving algorithm and might benefit from using formal notation like in [57]. This could also provide us with insight useful for optimizing the model weaver, and formally prove its correctness.

13.1.1 Auto-Generation of Weaving Points

Automatic generation of weaving points in primary interactions has until recently not been considered in AOMDF. A mechanism for automatically laying out weaving points (according to some rules or user input) in interactions is desired for the methodology to scale. Techniques for this are discussed in [43, 57].

13.2 COMPLETE MODELING TOOLKIT

In Figure 1-2 on page 3, we showed a conceptual proof-of-concept modeling toolkit for AOMDF. We have developed two of the four components shown there. Future work should have asone of its targetd to develop the other two components, graphical modeling tool and aspect repository. A suitable platform for quick relialization of the modeling toolkit will be EMF and GMF.

13.3 EXTEND AOMDF

Considering the broader perspective, AOMDF should be extended to support other UML2 diagram types. In our metamodel we have already included the *Structures*-package in *UML2SimpleMM*. By bringing in a few more of the UML2 concepts into *UML2SimpleMM::Structures* we can easily support modeling and weaving of composite structure diagrams.

Bibliography

[1] Simmonds, D., Reddy, R., France, R., Ghosh, S., and Solberg, A., ***An Aspect Oriented Model Driven Framework***, Proceedings of *Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05)*, pp. 119-130, Enschede, The Netherlands, 2005.

[2] Solberg, A., Simmonds, D., Reddy, R., Ghosh, S., and France, R., ***Using Aspect Oriented Techniques to Support Separation of Concerns in Model Driven Development***, Proceedings of *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, pp. 121-126, Edinburgh, Scotland, 2005.

[3] Reddy, R., Ghosh, S., France, R., Straw, G., Bieman, J. M., McEachen, N., Song, E., and Georg, G., ***Directives for Composing Aspect-Oriented Design Class Models***, *Transactions on Aspect-Oriented Software Development,* vol. 1, 1, pp. 75-105, February 2006.

[4] Reddy, R., Solberg, A., France, R., and Ghosh, S., ***Composing Sequence Models Using Tags***, in *Aspect Oriented Modeling Workshop at the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS / UML '06)*, Genova, Italy, 1-6 October, 2006.

[5] Solberg, A., Simmonds, D., Reddy, R., France, R., Ghosh, S., and Aagedal, J. Ø., ***Developing Distributed Services using an Aspect Oriented Model Driven Framework***, *International Journal of Cooperative Information Systems,* vol. 15, 4, pp. 535-564, December 2006.

[6] Elrad, T., Aldawud, O., and Bader, A., ***Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design***, in *Generative Programming and Component Engineering*, vol. 2487, pp. 189-201, London, UK: Springer-Verlag, 2002, ISBN: 3-540-44284-7.

[7] IBM Research, ***Multi-Dimensional Separation of Concerns: An Overview***, http://www.research.ibm.com/hyperspace/MDSOC.htm, Accessed: December 12, 2006

[8] Wikipedia, ***Separation of concerns***, http://en.wikipedia.org/wiki/Separation_of_concerns, Accessed: December 10, 2006

[9] Stuart, K., ***Model Driven Engineering***, in *Proceedings of the Third International Conference on Integrated Formal Methods*, pp. 286-298, London, UK: Spring-Verlag, 2002, ISBN: 3-540-43703-7.

[10] Greenfield, J., Short, K., Cook, S., and Stuart, K., ***Programming with Models***, in *Software Factories*, pp. 231-277, Indianapolis, IN, USA: Wiley Technology Publishing, 2004, ISBN: 0-471-20284-3.

[11] Mellor, S. J., Scott, K., Uhl, A., and Weise, D., ***MDA Distilled***, Addison-Wesley Professional, 2004, ISBN: 0-201-78891-8.

[12] Kleppe, A., Warmer, J., and Bast, W., ***MDA Explained***, Addison-Wesley Professional, 2005, ISBN: 0-321-19442-X.

[13] Schmidt, D. C., ***Model-Driven Engineering***, *IEEE Computer,* vol. 39, 2, pp. 25-31, February 2006.

[14] Filman, R. E., Elrad, T., Clarke, S., and Aksit, M., ***Aspect Oriented Software Development***, Addison-Wesley Professional, 2005, ISBN: 0-321-21976-7.

[15] Schauerhuber, A., Schwinger, W., Kapsammer, E., Retschitzegger, W., Wimmer, M., and Kappel, G., ***A Survey on Aspect-Oriented Modeling Approaches***, Technical Report, http://www.wit.at/people/schauerhuber/publications/aomSurvey, August 2006.

[16] Schauerhuber, A., Schwinger, W., Kapsammer, E., Retschitzegger, W., and Wimmer, M., ***Towards a Common Reference Architecture for Aspect-Oriented Modeling***, in *8th International Workshop on Aspect-Oriented Modeling (AOSD '06)*, Bonn, Germany, March 21, 2006.

[17] Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M., Bakker, J., Tekinerdogan, B., Clarke, S., and Jackson, A., ***Survey on Aspect-Oriented Analysis and Design Approaches***, AOSD-Europe (EU Network of Exellence), Technical Report, http://www.comp.lancs.ac.uk/computing/aod/papers/d11.pdf, May 2005.

[18] Rumbaugh, J., Jacobson, I., and Booch, G., ***The Unified Modeling Language Reference Manual***, 2 ed., Boston: Addison-Wesley, 2005, ISBN: 0321245628.

[19] The Object Management Group (OMG), ***Unified Modeling Language Specification: Superstructure. Version 2.0***, OMG, *http://www.omg.org*, August 2005.

[20] The Object Management Group (OMG), ***Unified Modeling Language Specification: Infrastructure. Version 2.0***, OMG, Final Adopted Specification, *http://www.omg.org*, March 2006.

[21] Pottinger, R. A. and Bernstein, P. A., ***Merging Models Based on Given Correspondences***, Proceedings of *29th International Conference on Very Large Databases (VLDB '03)*, pp. 862-873, Berlin, Germany, 2003.

[22] Gervais, M.-P., Engel, K.-D., Kolovos, D., Touzet, D., Shaham-Gafni, Y., Paige, R., and Aagedal, J. Ø., ***D1.5 Model Composition: Definition of Model Composition Properties***, Deliverabe in MODELWARE (EU FP6 IST Integrated Project 511731), *http://www.modelware-ist.org/*, 15 February 2006.

[23] Lundesgaard, S. A., Solberg, A., Oldevik, J., France, R., Aagedal, J. Ø., and Eliassen, F., ***Construction and Execution of Adaptable Applications Using an Aspect-Oriented and Model Driven Approach***, Proceedings of *7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2007)*, pp. 76-89, Paphos, Cyprus, 2007.

[24] Reddy, R., France, R., Ghosh, S., Fleury, F., and Baudry, B., ***Model Composition - A Signature Based Approach***, in *Aspect Oriented Modeling Workshop at the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS / UML '05)*, Montego Bay, Jamaica, October 2005.

[25] Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R., ***Computing as a Discipline***, *Communications of the ACM,* vol. 32, 1, pp. 9-23.

[26] Greenfield, J., Short, K., Cook, S., and Stuart, K., ***Language Anatomy***, in *Software Factories*, 278-320, Ed. Indianpolis, IN, USA: Wiley Technology Publishing, 2004, ISBN: 0-471-20284-3.

[27] Clark, T., Evans, A., Sammut, P., and Willans, J., ***Applied Metamodelling: A Foundation for Language Driven Development***, Xactium, 2004.

[28] Mens, T., Czarnecki, K., and Gorp, P. V., ***A Taxonomy of Model Transformation***, in *International Workshop on Graph and Model Transformation (GraMoT) at the Fourth International Conference on Generative Programming and Component Engineering (GPCE 2005)*, Tallinn, Estonia, September 28, 2005.

[29] ATLAS Group (INRIA & LINA), ***ATL (Atlas Transformation Language)***, http://www.eclipse.org/m2m/atl/, 2005.

[30] IBM Alphaworks, ***Model Transformation Framework***, http://www.alphaworks.ibm.com/tech/mtf, 2007.

[31] Sodifrance / Mia-Software, ***Mia-Transformation (in Mia-Studio Suite)***, http://www.mia-software.com, 2007.

[32] Triskell Team, ***Kermeta 0.4.1***, http://www.kermeta.org, 2007.

[33] Dijkstra, E. W., ***Selected Writings on Computing: A Personal Perspective***, pp. 60-66, New York: Springer-Verlag, 1982, ISBN: 0-387-90652-5.

[34] Mathiassen, L., Munk-Madsen, A., Nielsen, P. A., and Stage, J., ***Object Oriented Analysis & Design***, 1 ed., Aalborg: Marko Publishing, 2000, ISBN: 8777511506.

[35] Reenskaug, T., Wold, P., and Lehne, O. A., ***Working with Objects: The OOram Software Engineering Method***, Greenwich: Manning, 1996, ISBN: 0134529308.

[36] Pilone, D. and Pitman, N., ***UML 2.0 in a Nutshell***, O'Reilly Media Inc., 2005, ISBN: 0-596-00795-7.

[37] Warmer, J. and Kleppe, A., ***The Object Constraint Language***, Boston: Addison-Wesley, 2003, ISBN: 0321179366.

[38] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., ***Design Patterns: Elements of Reusable Object-Oriented Software***, Reading, Massachusetts: Addison-Wesley, 1995, ISBN: 0201633612.

[39] Luck, M., Ashri, R., and D'Inverno, M., ***Agent-Based Software Development***, Artech House Publishers, 2004, ISBN: 1-58053-605-0.

[40] Zimmermann, O., Kroghdal, P., and Clive, G., ***Elements of Service-Oriented Analysis and Design***, *http://www-128.ibm.com/developerworks/library/ws-soad1/*, Accessed: December 15, 2006

[41] Szyperski, C., ***Component Software: Beyong Object-Oriented Programming***, New York: ACM Press, 2002, ISBN: 0201745720.

[42] Pollice, G., ***A look at aspect-oriented programming***, *http://www-128.ibm.com/developerworks/rational/library/2782.html*, Accessed: December 16, 2005

[43] Solberg, A., Reddy, R., France, R., Ghosh, S., and Abbasi, M., ***An Approach to Composing Aspect-Oriented Design Models***, *(Draft, Planned for submission as a journal paper)*.

[44] France, R., Kim, D.-K., Song, E., and Ghosh, S., ***Metarole-Based Modeling Language (RBML) Specification. Version 1.0***, Colorado State University, Technical Report, 2002.

[45] France, R., Kim, D.-K., Ghosh, S., and Song, E., ***A UML-Based Pattern Specification Technique***, *IEEE Transactions on Software Engineering,* vol. 30, 3, pp. 193-206, March 2004.

[46] Kim, D.-K., France, R., Ghosh, S., and Song, E., ***A Role-Based Metamodelling Approach to Specifying Design Patterns***, Proceedings of *27th Annual International Computer Software and Applications Conference (COMPSAC 2003)*, pp. 452-459, Dallas, TX, USA, 2003.

[47] The Object Management Group (OMG), ***OMG MetaObject Facility website***, *http://www.omg.org/mof*.

[48] The Eclipse Foundation, ***Eclipse Modeling Framework website***, *http://www.eclipse.org/emf*.

[49] The Object Management Group (OMG), ***XMI Metadata Interchange***, *http://www.omg.org/technology/documents/formal/xmi.htm*.

[50] The Eclipse Foundation and SINTEF, ***MOFScript***, http://www.eclipse.org/gmt/mofscript, 2007.

[51] Fowler, M., ***Language Workbenches and Model Driven Architecture***, http://www.martinfowler.com/articles/mdaLanguageWorkbench.html, Accessed: December 25, 2006

[52] Gardner, T. and Yusuf, L., ***Explore Model-Driven Development (MDD) and related approaches: A Closer Look at Model-Driven Development and Other Industry Initiatives***, http://www.ibm.com/developerworks/library/ar-mdd3/, Accessed: May 15, 2007

[53] Solberg, A. and France, R., ***Navigating the MetaMuddle***, in *4th Workshop in Software Model Engineering (WiSME 2005) at ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005)*, Montego Bay, Jamaica, 3 October 2005.

[54] France, R., Ghosh, S., Dinh-Trong, T., and Solberg, A., ***Model-Driven Development Using UML 2.0: Promises and Pitfalls***, *IEEE Computer,* vol. 39, 2, pp. 59-66, February 2006.

[55] The TOPCASED collaboration, ***TOPCASED (Toolkit In Open source for Critical Applications & Systems Development)***, http://topcased-mm.gforge.enseeiht.fr/, http://www.topcased.org, 2007.

[56] The Eclipse Foundation, ***AspectJ***, http://www.eclipse.org/aspectj, 2007.

[57] Klein, J., Fleury, F., and Jèzèquel, J. M., ***Weaving Multiple Aspects in Sequence Diagrams***, *Transactions on Aspect-Oriented Software Development (Accepted To Appear)*.

APPENDICES

A Note on Delivered Artifacts

This appendix was meant to include source code listing and descriptions of the delivered artifacts. Unfortunately, a combined view of the metamodel, and complete listing of the model weaver's source code, is not suitable for paper-based presentation and will thus be made available upon direct request to the author or supervisor via e-mail. Please contact one of the following:

- Mansur Ali Abbasi (author), mansuraa@ifi.uio.no
- Arnor Solberg, (supervisor), arnor.solberg@sintef.no

Note also that these artifacts are the righteous property of SINTEF ICT, where the development took place. Fulfillment of any request is subject to individual consideration according to SINTEF's policies and classification of this material.